Identification And Simulation Of An Automated Guided Vehicle For Minimal Sensor Applications

A thesis presented for the degree of

Master of Engineering in Mechanical Engineering

in the University of Canterbury,

Christchurch, New Zealand.

By

Aaron Dann B.E (Mech) (Hons.)

1996

ABSTRACT

The problem of controlling an Automated Guided Vehicles (AGV) with the minimum number of sensors is considered. Sensors add cost and complexity to an AGV both electrically and in terms of increased computational requirements of the controller.

Computer simulations are proposed to model the behaviour of the AGV. Models of the dynamics of an AGV are proposed and simulated at varying levels of complexity using commercially available numerical software.

In order to model the AGV accurately, aspects of the control system and the physical system had to be analysed. Laboratory experiments were designed and performed, and the results were analysed to determine the dynamic properties of sub-systems of the AGV.

To provide a datum for comparison to the simulations, measurements were made of the performance of an AGV under a variety of control conditions corresponding to the computer models. Comparisons of the simulations and the AGV performance are discussed and suggestions are made for improving the AGV and its control system.

The models presented in this thesis demonstrate a good correlation for low performance AGVs in non-rigorous conditions, or well loaded AGVs on good traction surfaces. However they do not accurately represent the AGV at the limits of traction.

Two mechanical improvements to the University of Canterbury (UOC) Mk-II AGV are suggested, including the addition of softer compound tyres for use on hard, painted surfaces, and the design of a gear train with lower backlash.

TABLE OF CONTENTS

TABLE OF FIGURES

Acknowledgments

I am indebted to my supervisor Dr G.R Dunlop for his aid and encouragement during the course of this work.

I would like to thank the Applied Mechanics Laboratory Technician (and now PhD candidate) Andy Cree for his valuable aid in all things electrical and mechanical, and the Electrical and Computer Workshop Technician Julian Murphy for his help with things electronic.

I also wish to thank my fellow postgraduate students without whom the longest days would have seemed much longer.

I especially wish to thank my fiance Stephanie for her unbounded patience and support during the course of this work.

Glossary

AGV	Automated Guided Vehicle
A/D	Analog to Digital
ADC	Analog to Digital converter
ANN	Artificial Neural Network
DC	Direct Current Motor
DCMC	DC motor controller
DSP	Digital Signal Processor
EMF	Electro-motive force
EPROM	Electrically Programmable Read Only Memory
FET	Field Effect Transistor
FM	Frequency Modulation
IC	Integrated Circuit
IDE	Integrated Development Environment
I/O	Input/Output
LCD	Liquid Crystal Display
MCU	Microcontroller Unit
NZ	New Zealand
PC	Personal Computer
PSK	Phase Shift Keying
PWM	Pulse Width Modulation
UOC	University of Canterbury
UPP	Universal Pulse Processor

1. INTRODUCTION

The conventional pedestrian electric pallet truck is able to reverse under a load, lift it and then drive away following the walking operator. The development of accurate methods for controlling these vehicles has allowed the introduction of automated versions of these load carrying trucks. These automated versions are known as Automated Guided Vehicles or AGVs.

The University of Canterbury AGV MK II is an autonomous guided vehicle characterised by it's steering and driving axis geometry. The AGV achieves its steering by actuating the drive wheels at the different speeds, in much the same way as tracked vehicles.

This thesis presents the results of modelling the UOC AGV MK II in forward motion, with attention to options for driving the AGV in reverse. Mathematical models were developed to describe the dynamics of the AGV. These models were simulated using The MathWorks 'MATLAB' software package along with the SIMULINK simulation tools.

To ensure the accuracy of the AGV the simulation models, experiments were designed to determine the properties (co-efficients, moments of inertia etc..) of the AGV's sub-systems. The results from these experiments were analysed using the MATLAB and the System Identification Toolbox.

To validate the models the actual performance of the AGV needed to be compared to the results from the simulations. To ascertain this the UOC AGV MK II was equipped with a microcontroller based control system, sensors and electronic motor drive unit, and programmed to operate under the control methods used in the simulations. The AGV path was tracked and recorded . Finally the performance of the simulation was compared with the measured performance of the AGV.

1.1 ADVANTAGES OF AGVS OVER MANUAL LABOUR.

There are a number of important benefits to be gained from the use of AGV systems. Quite apart from the labour savings, these are mostly concerned with the transport properties:

(i) smooth regular acceleration and deceleration: Prevents product damage because the payload is not subject to the unnecessary stress of uncontrolled acceleration. Also the components of the AGV will have a greater lifetime as the stresses are known and controllable, and are not likely to exceed the rated design loads. This cannot always be said for manually driven fork lift trucks.

(ii) predictable (See (i) above) and controllable path, always travels by the shortest route: Although AGVs are not necessarily the fastest material handling method, they are the most flexible. They allow the product to be moved to the destination by the shortest available route at all times. However, this is subject to the effectiveness of the traffic management system.

(iii) regular service: The AGV does not make stops for the lunch, tea breaks etc...

(iv) minimal support services: The AGV does not require the creature comforts of human workers, heating, lighting, hygiene, facilities, superannuation etc...

1.2 OPERATION ENVIRONMENT CONSIDERATIONS:

AGVs will repetitively follow the allocated guide path, and because their wheels are generally hard on the work surface. The preferred surface is hard concrete since they will quickly wear tracks in soft surfaces such as carpet, vinyl or asphalt (tar).

In the case of wire guided AGV's, steel sheets or steel reinforcing mesh, on or near the surface can alter the signal from the guide wire. Appropriately spaced steel rods can however, be used as a path identification bar code.

Communication of AGVs with a central warehouse computer can be maintained by several methods:

(i) Radio Modem

(ii) Inductive Loop buried at decision intervals

(iii) Inductive wire buried alongside the guidance wire.

(iv) Physical connection data points

(v) FM or PSK (Phase Shift Keying) applied to the guidance wire.

The first method can suffer from the problems of multiple paths ie. interference patterns, if the systems are not properly set up. The radio signal can be reflected from multiple surfaces and in the high MHz frequency range, the wavelength (<1m) can give rise to antinodes in the work space where the radio signals suffer destructive interference thus leaving the AGV without any communication with the base computer. Fisher and Paykel NZ use AGVs in their refrigerator manufacturing plant as an integral feature of their flexible manufacturing system. These AGVs use radio modems to communicate with central routing computers and programmable controllers.

The second method has been used in a number of installations [13], however if the AGV stops away from the decision loop, it has no way to communicate with the central computer, and must simply wait until it is discovered. The third method is the most common, because the AGV follows the guidance wire and the adjacent induction wire gives immediate feedback to the central computer. A packet switching TDM can be incorporated when several AGV's are communicating on the same induction wire. Also, two signal pick-ups are required since the inductive wire position depends on which way the AGV is going . The fourth method is obvious but suffers from the same problems as method (ii).

The method (v) is proposed for the UOC (University Of Canterbury) AGV. The induction wire and the guidance wire are the same, which further reduces the capital investment and installation costs. In addition since the guidance pick-up is used for the communications signal as well, the two signal pick-ups required for (iii) are eliminated.

1.3 AGV MECHANICAL SYSTEM

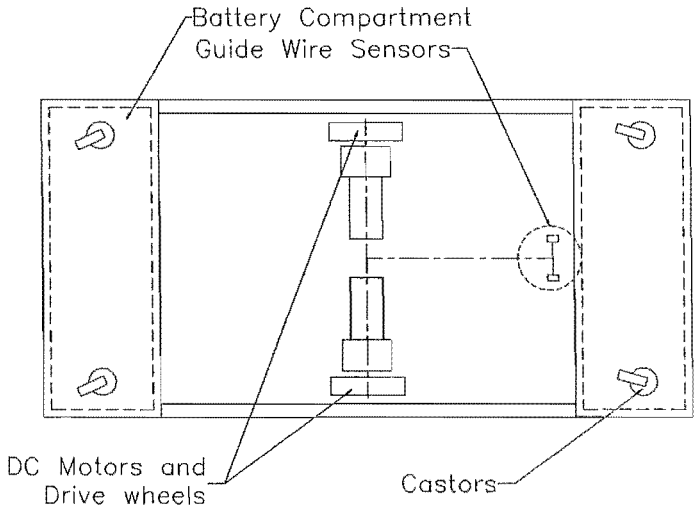


Fig. 1.1- AGV viewed from underside, note the battery compartments, guide wire antennae, and castors.

The UOC Mechanical Engineering School's 2nd AGV is a differentially driven unit in which each drive wheel operates to accomplish both drive and direction. The design payload of this vehicle is 200kg. This payload can be raised and lowered by means of a scissor-jack actuated platform. In terms of the overall control of the AGV, neither the platform nor it's moveability plays any part until cornering at speed. However the control system must able to provide for variations in the payload and the effect on the kinematics of the system (the change of linear and angular inertia).

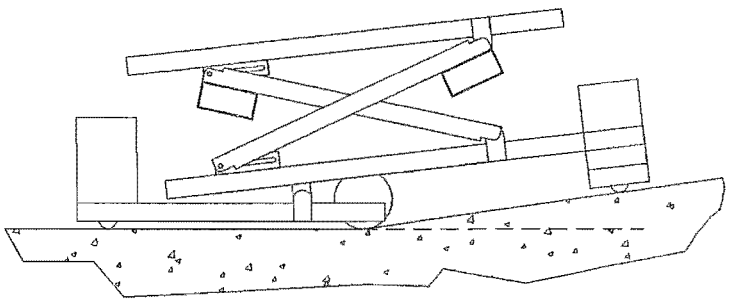


Fig 1.1- Side view of AGV mounting an inclined ramp

Mechanically the AGV is quite simple. An ingenious feature is that the two halves of the chassis are hinged to allow it to accommodate variations in floor height without the drive wheels losing contact with the ground and hence lose traction.

Drive is achieved by a two stage reduction from the DC motor through a ratio of 23.41:1. This ratio was chosen to give the AGV a maximum linear speed under its own power of average walking speed (approximately 1m/s), [18].

The first stage of speed reduction from the DC motors was by worm gearbox integral with the DC motor. The second stage of speed reduction was achieved by parallel axis gears. The worm drive had significant backlash. The complete gear train had approximately 10° deadband of rotation from the drive wheels to the DC motors. Running on the painted floor of the laboratory, the AGV was subject to slip on occasion when it attempted to change speed quickly.

Idler wheels (castors) under the four corners aided stability but added marginal extra weight and drag to the forward and rotational motion.

The chassis design allows space for four average sized automotive lead acid batteries to be stowed on board in compartments at the front and rear of the AGV. The guide wire antennae were mounted underneath an instrumentation tray, across the center line of the AGV.

The major mass/inertia properties of the AGV were calculated from the original CAD drawings. This gave values for angular inertia about the vertical axis, and overall mass, which were confirmed by weighing the completed AGV.

1.4 AGV ELECTRICAL SYSTEMS

The AGV is driven by two 250W permanent magnet DC motors as used primarily in wheelchairs. These electric motors are speed controlled by a proprietary motor control device designed and manufactured for wheelchair control by 'Dynamic Controls Ltd' of Christchurch NZ. This motor controller DC Motor Controller (DCMC) receives it's speed setpoints (velocity forward and turn velocity, Vf & Vt) from the AGV microcontroller which reads the position of the guide wire and applies control algorithms to calculate the Vf and Vt speed demands. Power for the electrical system comes from two (space is available for four) 12 Volt deep cycle lead acid batteries providing 24Volts.

Guidance is achieved by means of a wire loop at floor level (or buried in concrete in the case of an industrial installation), which transmits a low power (< 1 Watt) low

frequency (< 100 KHz) signal. Two tuned receivers attached to the underside of the AGV act as antennas. The AGV seeks to place the sensors either side of the guide wire. The differential signal from the sensors is used to determine the AGV centre line location with respect to the guide wire. The AGV's controller is an 8-bit embedded 80C552 microcontroller (a derivative of the Intel MCS51 family c.f. [24],[25]) with the on board facility for analog signal measurement and pulse timing generation and measurement. The microcontroller is coupled with an array of push buttons and an LCD panel for a user interface.

For programming testing and data collection, the AGV microcontroller connects to an IBM PC running DOS and a commercial 'C' programming suite for the 8051 family of microcontrollers. This enables the microcontroller to be programmed quickly using ANSI-C.

The System Identification input and output variables were obtained using a multipurpose I/O card containing analog inputs plus digital inputs and outputs (Universal Pulse Processor or UPP card designed and built by the Mechanical Engineering Department) which was installed in the host PC. This card measured timing pulses from the rotary encoders as well as the back EMF voltages for the AGV's electric motors. It also measured the demand voltages to the DCMC.

1.5 THE COMPLEXITY OF THE AGV SYSTEM

Minimising the number of sensors on an AGV should produce a cheaper and more robust product. Ultimately the control system could be reduced to an analog wire-guidance section supervised by a digital microcontroller for path planning and routing control. However the relatively low cost of microcontrollers with analog input channels, makes that simplification unnecessary. Also microcontrollers are generally more flexible than analog circuitry and the development of different control strategies is considerably easier.

In the case of the UOC AGV, there are eight analog input channels on the microcontroller, and the AGV uses nearly all these in its current configuration:

- 2 Analog inputs for guide wire sensors
- 2 Analog inputs for manual control joystick

- 1 Analog input for battery voltage level
- 1 Analog input for ultra-sonic distance measurement (if using a commercial integrated unit, otherwise it would be possible to achieve the same results with a digital output to trigger a transmitter, and a timer input to measure the echo).

An increase in the number of sensors is not a problem to the AGV microcontroller until the total number of sensors becomes greater that the number of available. In the case of the UOC AGV MK II, there are still two channels available. To use a number of sensors greater than those immediately available would require additional circuitry to read the signals, either multiplexing analog chips to select from the analog sensor signals, or additional analog to digital conversion chips (possibly memory mapped into the microcontroller).

Eight channel analog multiplexing chips cost from NZ$6 to $25 (single chip sales March '96), with 12 bit analog to digital conversion ICs costing from NZ$48 to $100 and above. There are a number of peripheral A/D converters on the market, including the Analog Devices, single chip serial A/D. These chips provide multiple channels of analog to digital conversion, the results of which can be accessed by the microcontroller via a serial protocol, these currently sell for around $25 in single units. The advantage of the serial chips is that they require little additional circuitry and often more chips can be networked to the microcontroller without difficulty.

When the price of additional analog inputs is compared to the price of the microcontroller, 80c552 (NZ$38), the cost of the microcontroller board is increased markedly with additional analog input channels above the existing eight.

1.6 ISSUES OF REVERSING

In AGV installations the necessity to reverse the AGV is most often limited to reversing out of docking stations where the distance to travel is short, and the well within the range of dead reckoning ,Imai et al.[13]. In the case where the AGV has only two guide wire sensors, its ability to reverse is limited as the AGV represents an unstable system with bounded ability to correct for disturbances.

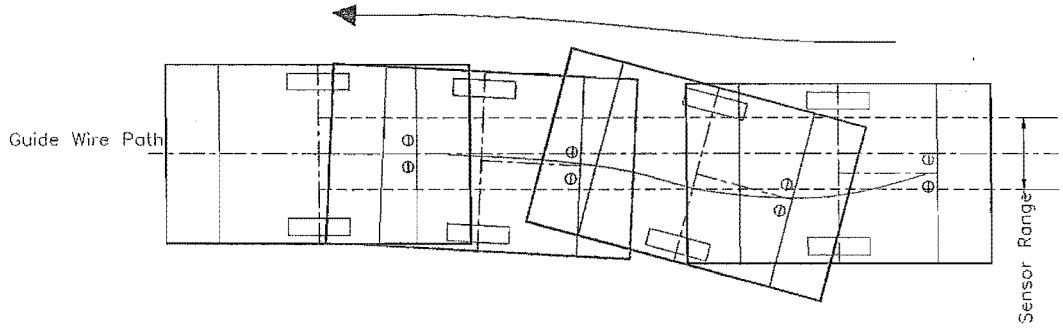


Fig. 1.1, Reversing the minimal AGV along a straight path from a start position offset from the guide wire.

When reversing the AGV along a straight path, it has to turn away from the guide wire in order to bring its central axis back into line with the guide wire. Depending on the magnitude of the displacement from the guide wire the AGV may have to rotate to a position where the sensors are no longer useful as the sensors have limited range. This puts the AGV where it cannot know its location with respect to the line without some memory of its past actions. In effect it requires the AGV to perform dead reckoning, where it postulates its position by integrating past actions. Without additional sensors to measure the distance travelled by the wheels or complex observer models, the AGV quickly becomes lost.

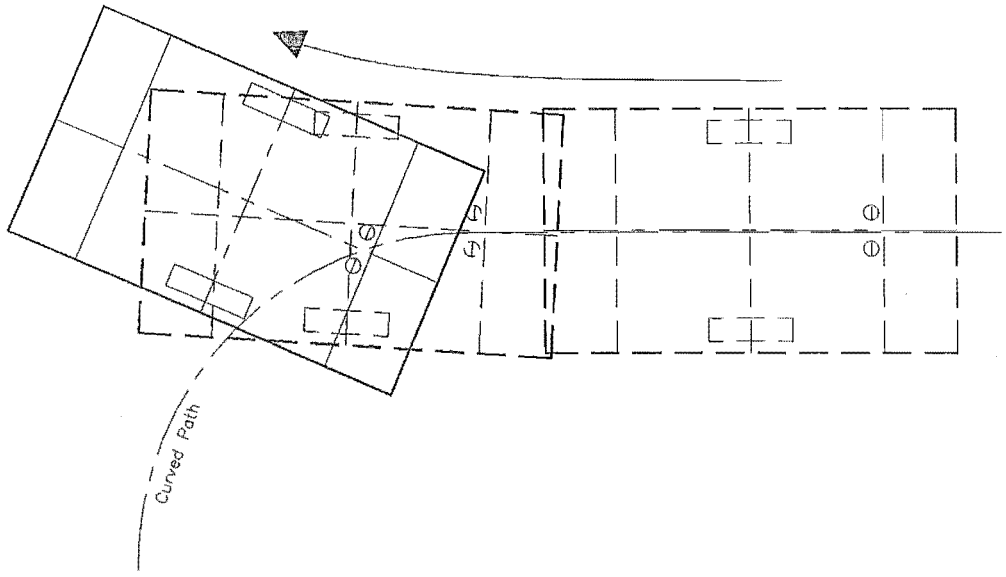


Fig. 1.2, Reversing the minimal sensing AGV along a curved Path

The reversing action is further limited to a straight line path. The assumption that a "swing away" control strategy makes is that it knows the location of the guide wire

with respect to the AGV, ie in a straight line. In the situation where the AGV is reversing along a path that curves, the minimal sensing AGV has no knowledge of the curve even after the centre of the AGV is well off the guide wire.

Adding more guide wire sensors to the UOC AGV can be handled a variety of ways depending on the arrangement of the sensors:

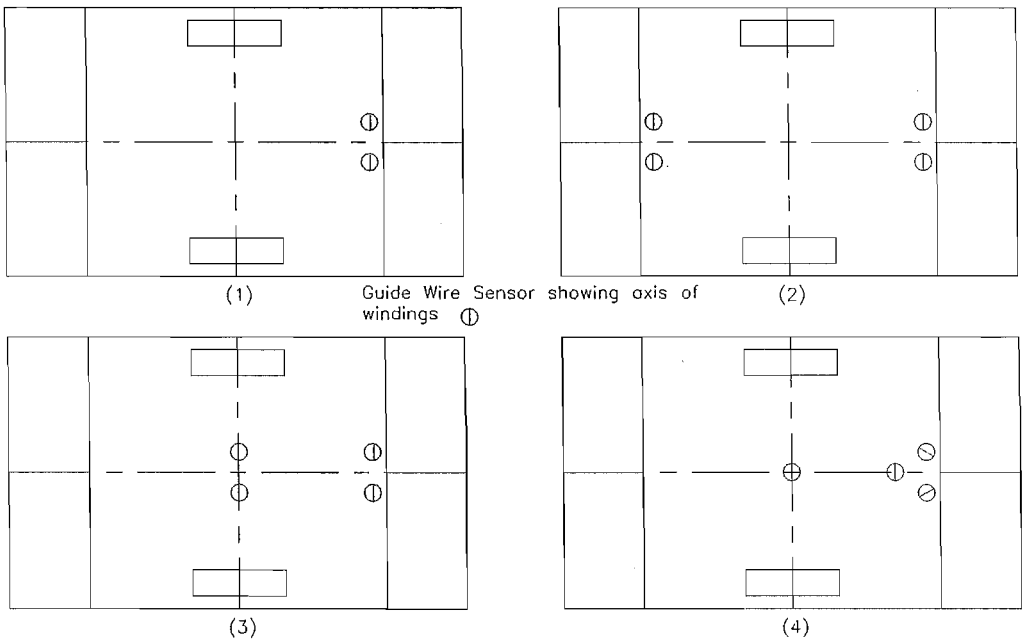


Fig 1.2, Various Guide Wire Sensor Arrangements for the AGV

In Fig 1.2, first diagram illustrates the existing sensor arrangement, the second case is the simplest option for providing reversing without complex changes to the guidance algorithms, the third and fourth diagrams show variations on the use of four guide wire sensors.

If in the second arrangement, the AGV desires to go backwards, it can switch to the guidance sensors at the opposite end of the vehicle. This would not require substantial changes in the AGV control algorithms, as such the effect on the computational load is negligible, and this is the easiest method for reversing the AGV.

In all cases but the first, more information could be gained about the position of the AGV from the additional sensors, including perhaps the approximate location of the centre of the AGV, or the location and orientation of the AGV with respect to the guide wire. The calculations to determine the AGV position data would be more

complex, as there are a greater number of variables to manipulate. The additional load on the microcontroller due to the extra variables would be dependant on the exact algorithm, but could be expected to increase the load by a factor of two to four by reason that each of the variables would have a third (and possible fourth) variable to interact with.

The strength of the sensor signal varies with the angle that the sensor makes with the guide wire such that when the sensor's axis is parallel to the guide wire the signal is zero. In the fourth illustration, the arrangement of the three sensors has an additional advantage over the arrays of pairs in the other illustrations, at least one of the sensors will always be able to detect the guide wire. This has the advantage that the AGV would be able to approach the guide wire from a more oblique angle without losing the signal, although intuitively, the algorithm to determine the AGV position and orientation would be quite complicated.

In short, to reverse the AGV under the minimal sensor criteria is difficult without additional sensors or control strategies. The most simple way around the problem of reversing the AGV is not to reverse it at all, but to add a second pair of sensors at the opposite end of the AGV and use them to drive the AGV forward in the opposite direction. The simulations in this thesis deal with developing a model of the UOC AGV MK II for the case of driving the AGV forward.

1.7 SAFETY CONCERNS

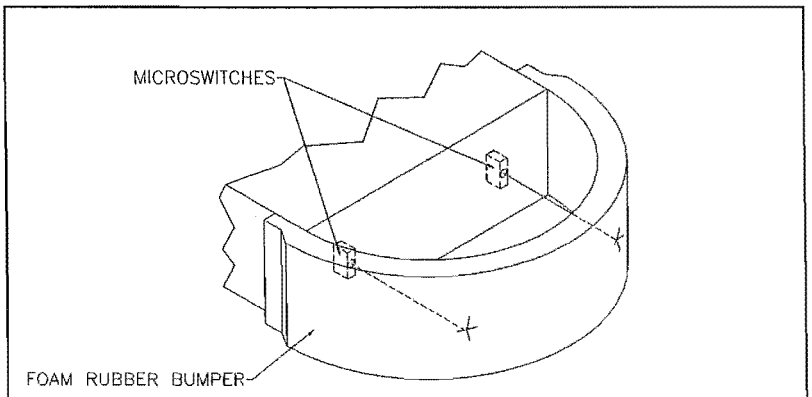


Fig. 1.3-AGV bumpers and microswitches

Front and rear bumpers were attached to the AGV to prevent damage to persons and the AGV alike. These bumpers were made of soft foam rubber bent around the front of

the AGV in such a fashion as to trigger the stop program on the microcontroller at the slightest touch of an obstacle.

The bumpers were designed to be fail-safe. The braid connecting the bumper and the microswitch was in tension and if it slackened for any reason, the microswitch would open with the loss of tension. The electrical circuit detecting the bumper was short circuit or 'normally closed' so any loss of power or a faulty connection would open the circuit and the microcontroller would cause the AGV to stop.

The distance the AGV travels from touching an obstacle to coming to a complete halt from loss of drive power was less than the depth of the bumpers. In this way the AGV would come to rest before exerting any substantial force on the obstacle. Stopping was achieved by removing the 'run' signal from the DCMC, which would engage a 'braking' mode to bring the AGV to a stop and attempt to hold the wheels stationary. This braked the AGV in a substantially shorter stopping distance than simply removing the power from the DC motors and leaving the AGV to coast to a stop.

NOTE: A peculiarity of the stop circuit was discovered during lab trials. If the bumper was depressed when the microcontroller was reset, the bumper would not trigger a stop during any subsequent collisions. This 'feature' of the microcontroller was related to the reset state of the external interrupt line that is triggered when the bumper is touched. However this was not well documented and remains a CAVEAT to future operators.

Other possible safety devices include sonar and laser/infra-red ranging systems for obstacle detection. These options require sophisticated signal processing and were not included on the UOC AGV unit. They would however be useful if the AGV could be operated as an autonomous unguided vehicle in some parts of a factory. The obstacle avoidance features could then be combined into a path finding algorithm.

2. AGV SYSTEM DYNAMICS

The various sub-systems making up of the AGV are examined in this chapter, and where the dynamics of a section cannot be accurately specified, system identification experiments were used to obtain the unknown model parameters. The performance of the complete system can be simulated for different control strategies. The most promising strategy can then be implemented and then the performance compared with the model predictions.

2.1 INTRODUCTION TO COMPUTER SIMULATION

The purpose of simulation is to attempt to model a system and its dynamics in order to evaluate potential control systems without having to implement the control system on the plant itself. This gives the control engineer the ability to manipulate and evaluate control system parameters without fear of damaging the actual plant.

Digital simulation of a system of a system modelled by dynamic equations is (in most cases) a numerical exercise. The computing power of modern desktop computers enables quite complex systems to be modelled with relative ease.

From the equations which define the interactions within the system being simulated, a model (or representation) of the dynamics can be determined. The linear equations of state can be represented in two ways:

(i) Difference equations

$$x(k+1) = f(x(k)) + g(u(k)) \tag{1}$$

These describe the next state in the time series, as a function of it's previous states and it's inputs. Difference equations are fast to compute on digital computers as the nature of the calculations lend themselves to high speed matrix manipulation (the realm of the digital signal processing chip or DSP). The sampled data state variable form of a linear system is:

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k \tag{2}$$

$$\mathbf{y}_{k+1} = \mathbf{C}\mathbf{x}_k + \mathbf{D}\mathbf{u}_k \tag{3}$$

This discrete time representation is particularly suitable for the Z-transform representation of a dynamic system. The sub-systems are easily combined in their Z-transform representations, and the results transformed back into a difference equation.

(ii) Differential equations

$$\frac{dx}{dt} = f(x(t)) + g(u(t)) \tag{4}$$

These describe the dynamics of a system where the equations represent a continuous time function. Differential equations are much slower to compute on a digital computer as the computer must 'extrapolate' states in time by calculating the rate of change of states and integrating the changes over time. The direct calculation of states by difference equations is usually faster.

The state variable form of a continuous time linear system is:

$$\dot{x} = \mathbf{A}x + \mathbf{B}u \tag{5}$$

$$y = \mathbf{C}x + \mathbf{D}u \tag{6}$$

To calculate the path of a group of states through time, from the differential equations, another means of calculation must be invoked ie. numerical integration algorithms. The equations of motion (or state variable matrices) of a system describe the rate of change of states, at a given instant. Approximations must be made as to the system's state in the future. These are based upon the derivatives computed from the state equations, and also on the past states in the case of some algorithms which fit polynomials to the state variables.

In 'Simulink' for Matlab™, there are a number of possible integration algorithms to choose from including;

- 3rd order Runge-Kutta: A 3rd order polynomial is fitted to the state paths. The algorithm is known for its ability to 'self start' from a position where there are no previous states calculated. It fits the polynomial to the derivatives and previous states of the system. Implementations of other integrators have been known to use a Runge-Kutta integrator to calculate the first few steps of a calculation to start their algorithm from.

- 5^{th} order Runge-Kutta: A 5^{th} order polynomial is fitted to the state paths. This algorithm has the same features as the 3^{rd} order Runge-Kutta but uses the higher order polynomial fit.

- Euler: This simple method uses the state equations in the first two terms of the Taylor series:

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{7}$$

  As the Euler method is a relatively fast algorithm to calculate, it was typically used to perform a brief trial simulation of a model to ascertain the stability and validity of the numerical model before performing a simulation using a more complex (and slower) algorithm such as the 3^{rd} order Runge-Kutta.

- Adams-Gear: A predictor-corrector algorithm which uses two previous time steps to calculate a polynomial fit to the state paths.

Each integrator has it's peculiarities and sensitivities to the system of equations it is being used to solve. With some models, the results for a known stable system can become quite badly unstable forcing the supervisory software to halt the simulation. This may be as a result of poor choice of time step, or because the integrator has too great (or small) a range of time steps to operate within. Computation time increases with O(1/n), where n is the time step, so n is usually made a large as possible and model is designed to minimise needless calculation.

Choosing appropriate time steps can be left to the integration algorithm. The algorithms used by Simulink have the advantage of incorporating adaptive time steps which adjust with the rates of change of the system being simulated. If the system states change quickly with time, the algorithm chooses a smaller time step to increase the resolution at that time. Conversely it will increase the time step if the system changes slowly with time.

After determining the system's equations of motion the plant model can be represented by its equations of state. These can also be converted to transfer functions where required.

2.2 DERIVATION OF THE AGV MODEL

In order to simulate the AGV system, a numerical representation or 'model' had to be created that represented the dynamics of the AGV. This section describes the process of developing such a model of the AGV.

2.2.1 Notation

(i) AGV System Variables

M_{AGV} = Complete mass of the AGV including motors, batteries etc. (kg) 8)

J_{AGV} = Angular moment of inertia of the AGV about the vertical axis through the centre of the differential steering axes. ($kg \cdot m^2$) 9)

J_{DC} = Angular moment of inertia of the DC motor rotor. ($kg \cdot m^2$) (10)

K_V = DC motor Back EMF coefficient. $(\frac{Volt \cdot seconds}{rad})$ (11)

K_B = DC motor Torque coefficient. $(\frac{N \cdot m}{Amp})$ (12)

R_C = DC motor armature resistance (Ohms)s (13)

N = Gear ratio from DC motor to drive wheels (14)

R_w = Diameter of the drive wheel (15)

D_W = Drive wheel separation (16)

C_{AGV} = Linear viscous damping coefficient (17)

B_{AGV} = Angular viscous damping coefficient. (18)

I_{tot} = Linear inertia of the AGV including the Mass and referred polar inertia of the DC motors (19)

$$= M_{AGV} + \left(\frac{N}{R_w}\right)^2 J_{DC}$$

J_{tot} = Linear inertia of the AGV including the Mass and referred polar inertia of the DC motors (20)

$$= J_{AGV} + 2\left(\frac{D_W N}{2R_w}\right)^2 J_{DC}$$

(i) AGV State Variables

θ_{AGV}, $\dot{\theta}_{AGV}$, $\ddot{\theta}_{AGV}$ = Bearing angle from the X axis, angular velocity and acceleration of AGV about the centre of mass. (21)

V_{AGV}, $\dot{V}_{AGV}$ = Forward velocity and forward acceleration of the AGV's centre of mass. (22)

$\dot{\theta}_{L,R}$, $\ddot{\theta}_{L,R}$ = Angular velocity and acceleration of the left and right DC motors respectively. (23)

2.2.2 Introduction

In most mechanical systems it is possible to describe the system dynamics in terms of power/energy flow between the component sections. The AGV is no exception.

Power bond graphs [22] are a convenient method for describing a mechanical system and give a useful guide for direct derivation of the state equations. Each bond describes energy flow from one component to another, thus indicating the causality and direction of effort of the interaction.

Traditional 'Effort/Flow' pairs are Voltage/Current, Pressure/Flow, Force/Velocity, all of which describe classic Action-Reaction pairs; it is worth noting that of these traditional pairs, the flow is a time integral of the acting effort. This is convenient for the state space representation of dynamic systems.

On the AGV a simplified Bond graph appears as follows with energy being passed from the motors to the gear train to the AGV chassis. The efforts are the voltages applied to the DC motors and the torques generated by the motors. The flows are the electric currents and the motions of the components of the AGV.

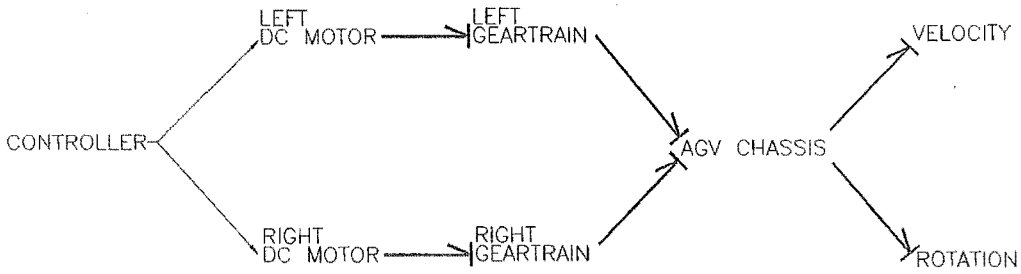


Fig. 2.4- Energy flow in a simplified bond graph of the AGV

The effort/flow duality is particularly convenient when describing the motion of simple mechanical components. In particular the classical energy transfer analogies for inertia, spring, and viscous damping components are used ie. energy flow as kinetic and potential storage, and dissipative respectively.

Using the bond graph, the systems for the block diagram of the AGV can be readily identified as follows.

2.2.3 The AGV's Equations Of Motion

The equations of motion of the AGV turn out to be relatively simple. The AGV can be treated as a "Lump-Mass" with two variable (controllable) forces applied to the mass at the contact point of the wheels. Resisting the applied control forces are internal forces due to viscous damping in the gearboxes, and friction due to the motion of the idler wheels.

2.2.4 AGV Sub-systems

The AGV is treated as a collection sub components and the relationships between the sub components are described mathematically.

DC motor

Using the following model for a DC motor,

$$V = Ri + L\frac{di}{dt} + k\omega \tag{24}$$

$$\tau = ki \tag{25}$$

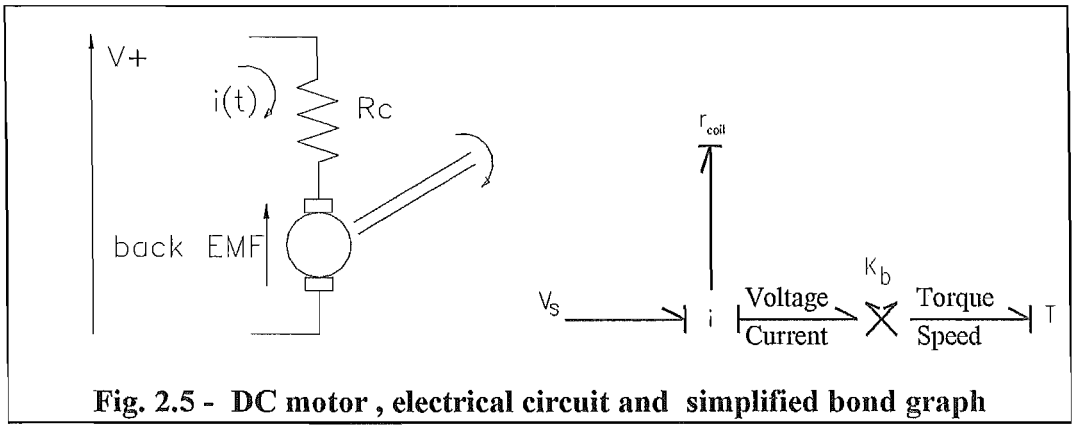


Fig. 2.5 - DC motor , electrical circuit and simplified bond graph

Which is equivalent to the following block diagram:

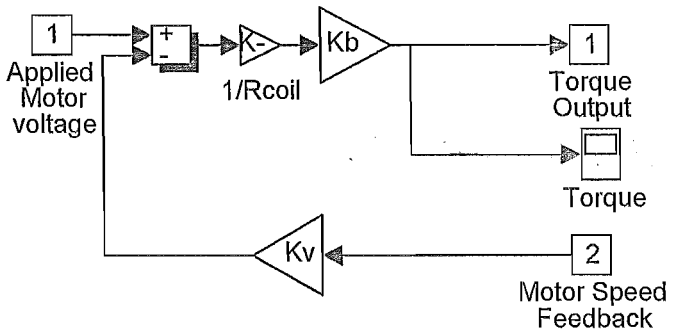


Fig. 2.6- DC Motor Simulink Block diagram

Note the similarity between the effort/flow duality in the bond diagram and the arrangement of the connections in the block diagram. Using this method of interfacing between sections of the model simplifies model development considerably.

Reduction Gear Train

Primary terms in the equations of motion of the drive train are the behaviour of the DC motor along with the rotational inertia and the viscous plus friction damping in the gearbox.

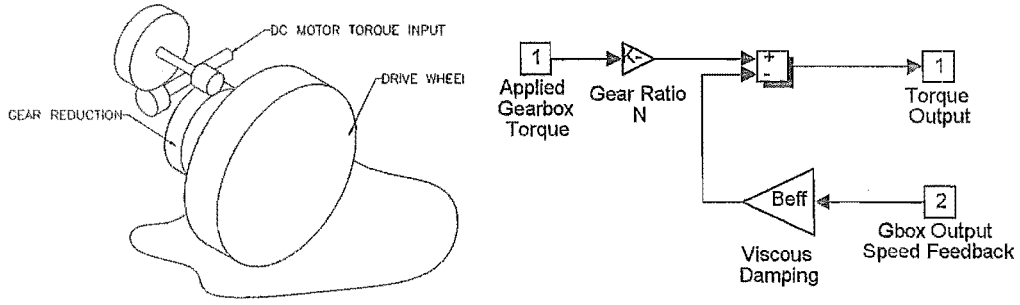


Fig. 2.7 Reduction Gear Train, Mechanical Representation and Block Diagram

AGV Chassis

For the AGV itself, the efforts are the forces applied at the wheels and the flows are the wheel velocity. The motion of the centre of mass of the AGV is the primary concern of the control system when guiding the AGV around a path.

The Idler Wheels (Castors)

These provide stability for the AGV chassis by holding up each end of the AGV, but they add additional terms which increase the complexity of the model. Each idler wheel is a castor which is free to rotate around a vertical axis which is eccentric to its contact patch. This can lead to the "Supermarket Trolley Effect" where the idler wheels can induce unstable oscillations into the motion of the AGV at high speed. Fortunately the AGV was not operated at high speed. B. W. Johnson et al [10] found that the contribution of the idler wheels to the kinematics of the AGV is negligible, and that they were equivalent to a component of the viscous damping of the AGV in the forward and rotational sense. Thus the drag component of the idler wheels was ignored and assumed to be combined with the overall AGV friction damping terms.

2.2.5 Model Development

As with all simulations, the systems of equations determining the motion of the AGV can be examined at any degree of complexity from the inane to the excruciatingly

trivial. Relevant non-linearities were included as the model was developed from the most simple case.

Using only integration operations wherever possible reduces numerical stability problems when computing the system dynamics. Numerically calculated derivatives can cause instabilities in the solution if the time steps are not appropriately chosen. Also, the use of the 1/s operator maintains consistency with the state variable approach to matrix representation of the AGV model.

In previous sections the inertia of the gear train and the DC motor were neglected. In order to maintain a standard of using only integration operators (1/s operators) within the model, the inertia terms were best collected as 'referred' inertias to the reference frame of the AGV chassis.

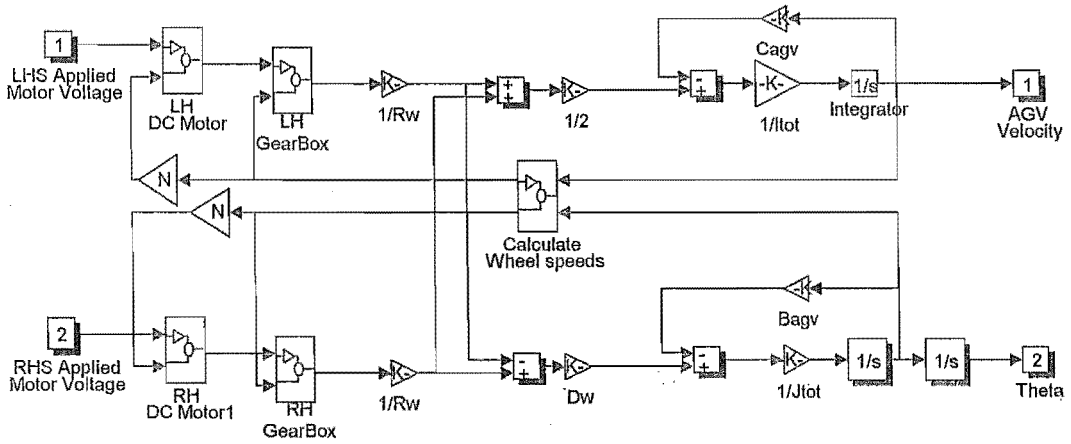


Fig. 2.8- Complete AGV Block diagram

From the models of the individual components, the complete dynamic model of the AGV can now be assembled.

The following assumptions are contained in the model

1) The drive wheels of the AGV do not slip.

2) The centre of gravity of the AGV is at the center of the wheelbase.

3) All damping is viscous in nature, no 'stiction' or Coulomb friction is included.

4) No backlash in the gears of the model, even though the AGV gear train described significant backlash (10° at the wheels).

2.3 Determining System Properties Using System Identification Methods

Within the dynamics of the AGV were a number of sub-systems with unknown co-efficients and properties. These included, the behaviour of the DC motor, and most importantly, the behaviour of the DCMC. In order to control the AGV properly it would be preferable to know the dynamics of these sub systems, but determining these dynamics is an interesting problem. One approach would be to give the AGV known inputs and determine the order and amplitude of the dynamics by examination, relying on the experience of the observer to estimate the dynamics implicitly.

However rather than rely solely on the interpretation of the results by an observer, it was decided to attempt to determine these sub-systems by means of systems identification techniques.

The problem of system identification is to estimate a model of a system based on input and output data, and a priori knowledge of the possible nature of the dynamics of the system. As the DCMC, and the dynamics of the power transmission system were the main unknown areas of the AGV, they were investigated using the following methods.

The core of the System Identification Toolbox is the representation of a model, or system of equations, as a discrete time sequence. The Matlab tool used in this analysis was the least squares error approximation tool (PEM.M). A complete description of the parameter process can be found in L. Ljung[16].

The observer treats the unknown system as a 'black box' (a convenient notation in the case of the DCMC as it was a black box), feeds measured demands to the system, and records the outputs. The Matlab 'System ID' toolbox takes care of most of the computational detail, and the observer is free to examine a number of proposed models. The problem can them become one of model reduction, or what order of model best represents the dynamics being examined.

The solution of a system model is an iterative process involving the presentation, calculation and evaluation of candidate models based on the systems I/O data gained from experiments. Experiments must be appropriately designed (where possible) to isolate or exaggerate the desired dynamic effects.

2.4 THE EXPERIMENTS:

A number of experiments were performed in order to isolate the dynamics of the target systems. In the case of the AGV , it is relatively simple to isolate the sub-systems from the AGV and examine them individually. In a more complex system this would not necessarily be possible.

2.4.1 Recording the dynamics of the DCMC and gear train

The AGV was raised off the ground, allowing the drive wheels to rotate freely. The DCMC input section was attached to the microcontroller.

In all cases a Universal Pulse Processor (UPP) card (Hitachi HD64130) was used in the host PC to measure both the analog demand signals from the microcontroller, and the speed of the DC motors which was inferred from attached position encoders.

The following methods were used to generate the demand signals:

1. A program written in C running on the microcontroller was used to generate signals of pseudo-random pulse length and intervals. These ±100% demand signals were sent to the DCMC.
2. Used a free standing joystick to generate a random input pattern to the DCMC, the operator moved the joystick in the most random manner that could be managed.

Test runs lasted from 2-15 seconds and the results of which were recorded in the memory of the a host PC. After the test was completed, the data was written to a file for later analysis with Matlab.

From the presentation of the data of the tests above, a number of models were formulated for the dynamics of the DCMC.

(1) The first proposed model for the DCMC is a simple gain matrix taking two signals representing forward and turn demand, and converting them into left and right wheel speeds by varying the voltage applied to the wheels. Simple DC motor theory will set the wheel speed to a given value for a voltage (at a given load).

(2) The second model presumes that the DCMC has some measure of feedback logic and can vary the output voltage to achieve the required wheel speed. The exact measure of this feedback is unknown so two possibilities are proposed:

I. Simple Proportional gain

II. Proportional + Integral gain.

From the results obtained, the DCMC could incorporate either model (I) or (II), as there was no compelling evidence for a more complex control strategy being used in the DCMC.

NOTE: The experiments were not originally designed to perform the analysis of the second option. To obtain better results in determining the operation of the DCMC, another scheme would be needed. Using an intermediate voltage measurement level it is then possible to determine the voltage applied to the motors by the DCMC, rather than attempting to imply the operation of the DCMC from the motion of the wheels.

2.4.2 Recording and analysis of the Electric Motors and drive train

From the equations of motion of a simple DC motor, the dynamics of an electric motor are determined by the following co-efficients.

K_v = coefficient of proportionality of back - EMF from shaft rotational speed (26)

L_c = DC motor coil inductance (27)

R_c = DC motor armature resistance (28)

J_{DC} = polar moment of inertia of the rotor (29)

Certain constants in the drive motors were unknown, the inertia and viscous friction of the DC motor rotor, the brush friction and the Kv constant of proportionality for calculating the back EMF from the motor speed. With the exception of the motor coil inductance, these values are quite crucial to the AGV model dynamics.

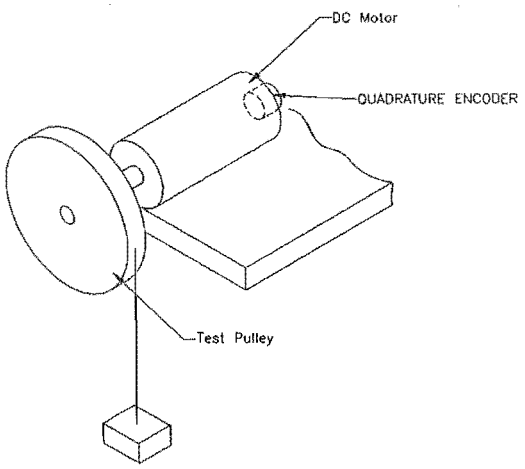


Fig. 2.9- Back EMF, DC motor rotor inertia test rig.

In order to find the polar moment of inertia of the DC drive motors, they were removed from the AGV and quadrature rotary encoders were attached to the output shaft. A pulley was attached to the motor output shaft and a weight was hung off the pulley at table height.

The weight was dropped from table height, and the UPP card was used to measure the timing pulses from the encoders, and the back EMF generated across the motor terminals at speed. A number of different weights were used. This data was recorded and saved to file for later analysis.

To attempt to measure the viscous damping of the DC motors rotor, the AGV was lifted off the ground (fully assembled) and the electric motors were driven at full speed by connecting the motor terminals directly across the drive batteries. With the motor at full speed, the power was removed, and the wheels were left to coast to a stop under the effects of whatever damping was present. The motor speed and input voltage were measured and recorded with time.

2.4.3 Recording the performance of the AGV under closed loop control

Measuring the performance of the AGV, the data of interest is the path of the AGV with respect to the guide wire during closed loop control. A number of methods were considered to measure the actual performance of the AGV under closed loop control. These included:

I. Data logging from extra sensors to perform 'dead-reckoning'. For example adding position encoders to each wheel to measure the speed, and calculating the position of the AGV over time by integrating the encoder records.

II. By attaching cables to stationary points, and calculating the position of the AGV from triangulation of the cable lengths to the AGV. This method could also provide an instantaneous direction vector by adding a position encoder to one of the cables at the AGV.

III. Direct measurement of the AGV position by means of a pigment trace on paper. (effectively using the AGV as a 120kg 'Logo' turtle)

Method I would suffer from integration and quantisation errors over time. Imai et al [13] state that open ended dead reckoning (without any means of calibration with the real world position) has limited range, and becomes less accurate with the increasing distance travelled. Also, dead reckoning is dependant on a no-slip condition ie. constant wheel speed with AGV speed, or complex modelling of the slip functions, c.f. A. Hamdy and E. Badreddin [5]. Wheel slip will introduce errors into the position calculation that would be magnified with subsequent readings. Even though the AGV models assume no-slip at the drive wheels (in fact wheel slip was observed on occasion). There is no way to guarantee this in the actual AGV without additional engineering ie. adding position sensors to the idler wheels. This reliance on the no-slip condition meant that the dead reckoning method was unacceptable.

Method II was too complicated and costly to implement when compared with the other options.

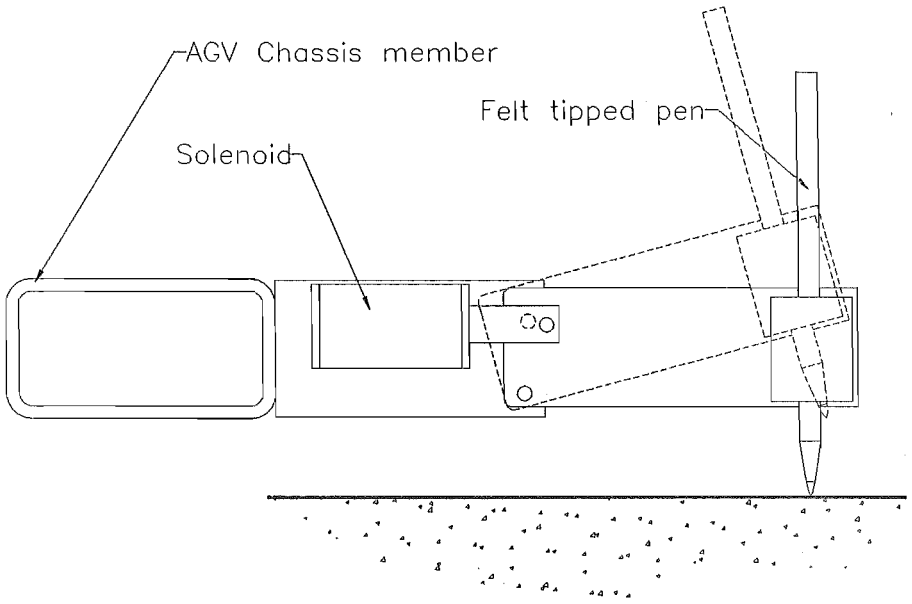


Fig. 2.10- AGV path marking pen

Method III was chosen for its immediate applicability, the paper and pen method provides the most accurate representation of the AGV performance, irrespective of wheel slip.

A solenoid actuator was mounted under the center of the AGV. A felt pen was attached to the solenoid via a bell crank. When the solenoid actuated, the pen was lifted free of the paper. A bi-stable oscillator drove the solenoid via a FET circuit. The solenoid would toggle on and off at a set rate according to the frequency of the oscillator, hence producing a series of dashes on the paper laid along the path of the AGV.

To perform a test run, the guide wire signal generator was turned on and the guide path was checked. Paper was laid down over the guide wire and fixed to the floor. After the position of the wire was marked on the paper, the AGV was moved into position (taking care not to rip the paper). The AGV was turned on and the test parameters were entered into the user panel. The microcontroller program waited until a key was pressed on the user panel. At this point the solenoid actuated pen was turned on and after it was verified operating, a key on the user panel was pressed. Upon a button press, the microcontroller would pause 2 seconds, engage the DCMC, pause again, and engage the control algorithm to be tested, setting off to leave a series of dashes on the paper behind the AGV.

2.5 IMPLEMENTATION OF THE AGV MODEL

The Simulation was performed using MATLAB-PC on various '486 computers running Windows 3.11. The target system's transfer functions were entered as a combination of functions in 'M'-file format (Matlab's programmable macro file format) and as Matlab 'Simulink' models in order to perform the simulation.

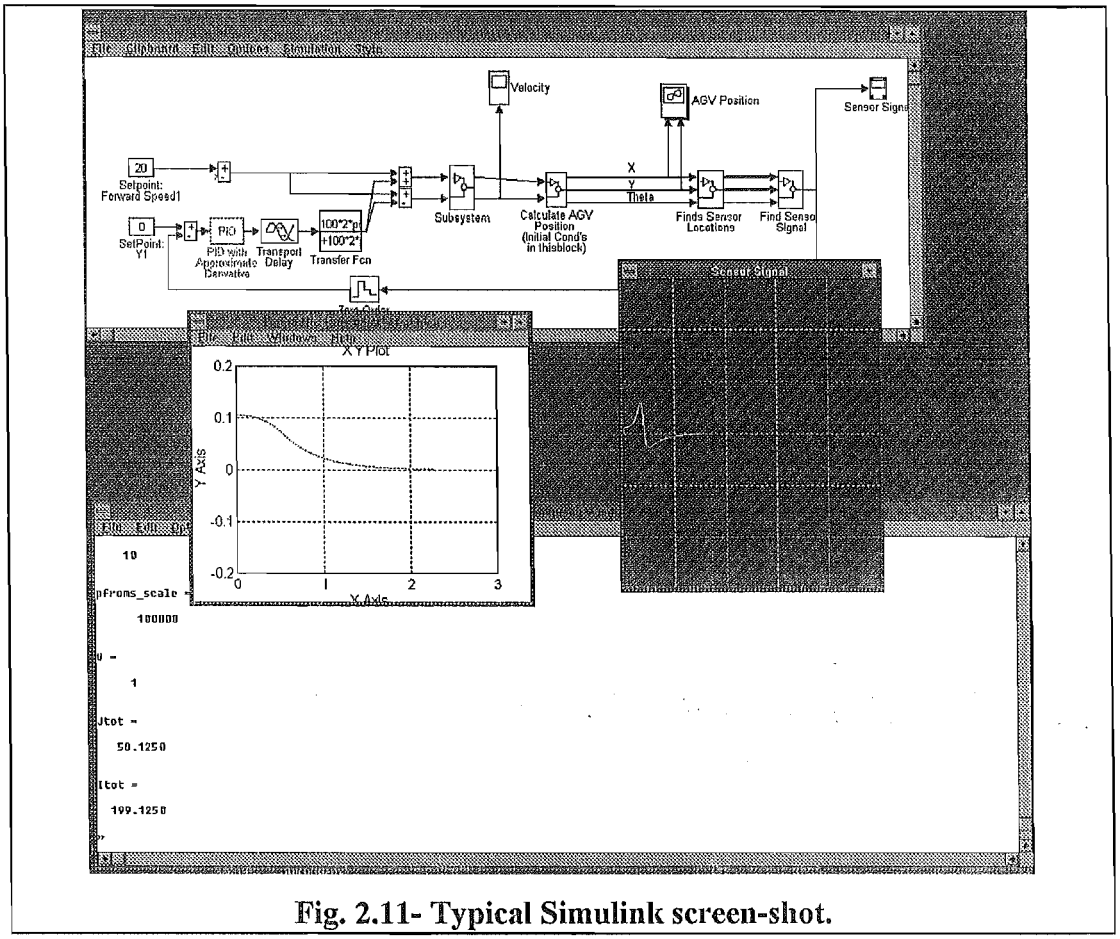


Fig. 2.11- Typical Simulink screen-shot.

'Simulink' is a graphical user interface to Matlab's powerful simulation functions. It uses block diagrams as one of its principle interface features to graphically represent a dynamic system which would previously have been described in terms of its state equations. The difference in the computational speed of a single M-File and an equivalent 'Simulink' model could be of the order of three to six times slower for 'Simulink' models. However the graphical environment used for Matlab 'Simulink' has the enormous advantage of simplicity of model interpretation and debugging. The ability to attach a 'virtual-scope' to any connection on the diagram, and to examine the changing values, is extremely useful for debugging and for understanding the inner workings of a dynamic system.

The 'Simulink' model of the AGV was developed through the following stages of complexity.

2.5.1 Non-Linear - Second Order Model

- Saturating DC motors.

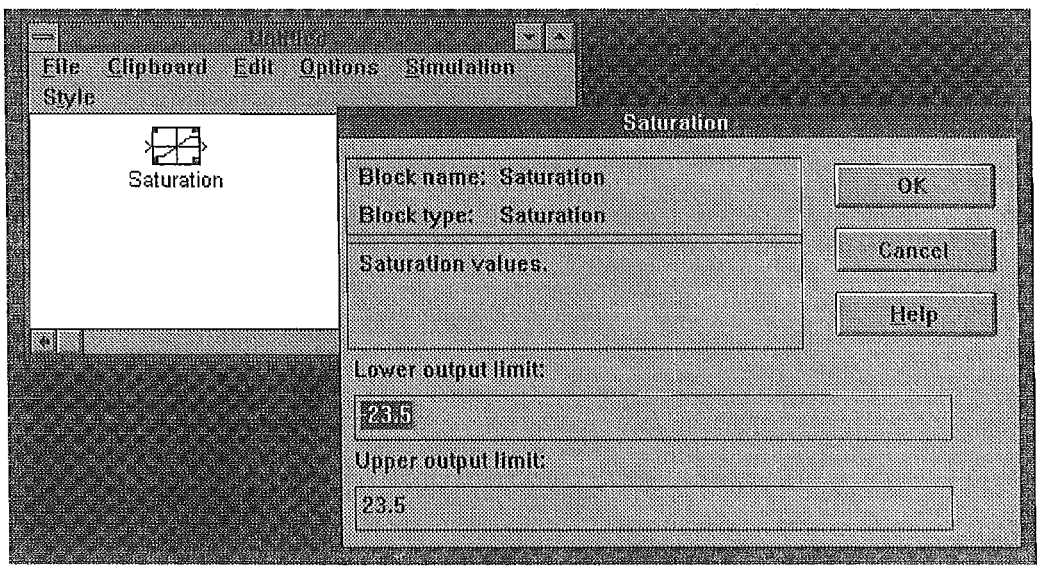


Fig. 2.12- Simulink Saturation Block, ±23.5Vdc

The voltage available to drive the motors is limited to ±24Vdc, which is the total battery voltage available from the lead acid batteries. Actually, the available voltage varied according to the current drawn (due to the internal resistance of the batteries), and would vary from 24Vdc at standstill to 23.5Vdc under full load. Also the voltage varied with the charge state of the batteries. As the battery discharged, the available voltage dropped. For the purposes of the simulation the saturation voltage was set to 23.5Vdc. A low battery voltage (22Vdc) would cause the DCMC to fault (ie. to halt the AGV) indicating this low voltage fault on the status LED.

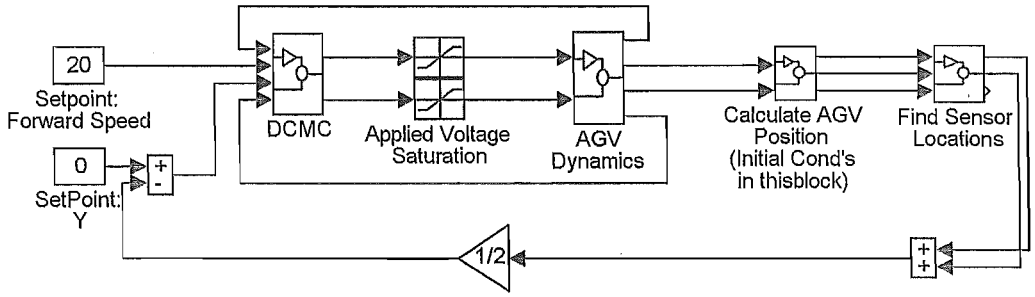


Fig. 2.13 - Simulink Screen: Simple Non-Linear AGV Model

- Linear sensor functions

For this model, the AGV was assumed to have linear feedback of the location of the sensors. This would have been the case if some means of linearising the guide wire sensor data had been successfully implemented.

- True 2-D Position calculation Position Calculation Block

The AGV dynamic equations were formulated in co-ordinate systems referred to the AGV, (Vagv, ThetaAgv). In order to view the performance of the AGV in a 'real world reference fra e, the position of the A is calculated by the following bloc .

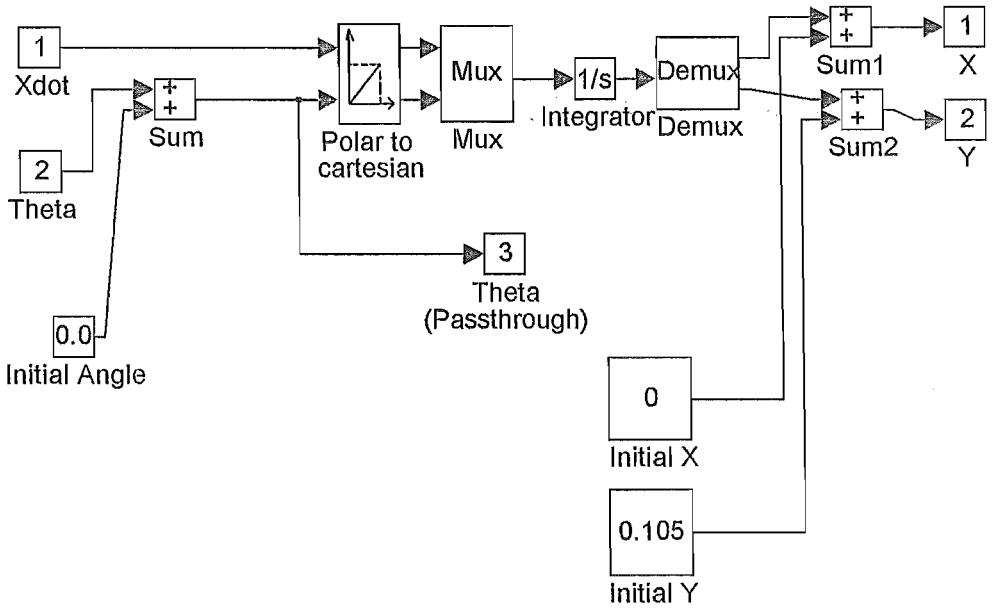


Fig. 2.14- Simulink Screen: Position calculation block

The block calculates the following equations numerically to determine the path of the AGV.

$$x = x_0 + \int_0^T V \cos(\theta + \theta_0) \cdot dt \tag{30}$$

$$y = y_0 + \int_0^T V \sin(\theta + \theta_0) \cdot dt \tag{31}$$

where (x_0, y_0, θ_0) are the initial conditions:
$\dot{\theta} = V = 0$, at $T = 0$.

2.5.2 Complete Simple non-linear model

- Non-linear sensors

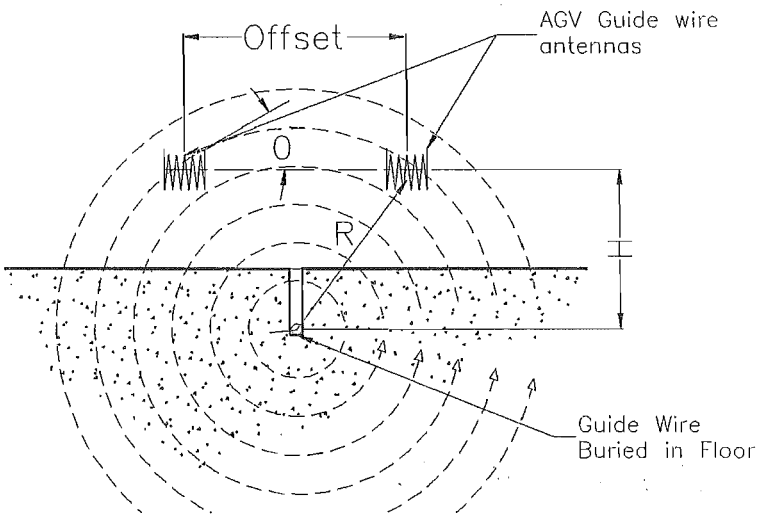


Fig. 2.15- Guide Wire Sensor, sensor strength diagram

The non-linear sensor block is a function that is designed to have outputs that match the outputs of the guide wire sensor circuit for a given position of the AGV. The circuit has reception properties determined by the location of the sensor relative to the guide wire. The signal strength at the sensor is proportional to the inverse of the radial distance to the sensor, and the cosine of the angle that the sensor axis makes with the field.

The simulation block calculates the location of the sensors from the position of the guide wire and uses these locations to determine the sensor signal levels that would be present.

Originally the resolution of the sensor locations was to be performed by either an Artificial Neural Network (ANN) or by a look-up table. Upon formulating and

training (off line) a neural network to perform the function inversion, it was found that the computational overhead for a Neural Network on the given microcontroller was such that it would only be capable of performing a position calculation 4 times every second. This was unacceptable, and rather than re-code it to some faster method (it involved re-writing large sections of the floating point math library in either fixed point or integer math) it was decided to discard this method. Upon closer examination of the sensor function, there is more than one point on the curve where the sensor values are the same for different locations. Thus a look-up table was not very reliable.

- Time delay on 1st Order S&H, time delay on calculation of control value

In order to simulate the control system more closely, the microcontroller's sample and hold operation was implemented. The microcontroller used a timed interrupt to perform the control algorithm, operating as a discrete time controller. Thus it has a limited resolution. Using a linear model is not entirely appropriate as it has effectively infinite resolution (only limited by the time step of the simulation). The sample and hold block simulates the operation of the process of sampling the A/D converter, and storing the sensor values. The microcontroller then operates on those values, and presents the new speed demands to the DCMC by over writing the old values.

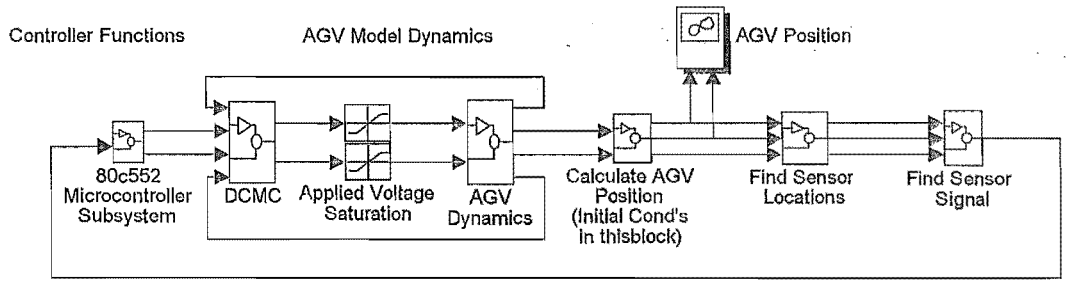


Fig. 2.16 - Simulink Screen: Complete Simple Non-Linear Model

2.5.3 Complete Complex Model - second order AGV + backlash

The complete complex model included the following:

- Limited sensor Resolution, using the quantisation block from 'Simulink'
- Backlash in the drive train

Backlash was evident in the drive train of the AGV on early inspection. A backlash model requires splitting the simple model into three parts and re-defining the interaction between the AGV chassis, the DC motors and the controllers to include the effects of backlash in the gear train.

Changes were made to the model so as to treat the drive trains, and the AGV chassis as separate entities joined by the interaction of forces. The magnitude of these forces is defined by the relative position of the gear teeth in the gear train under load.

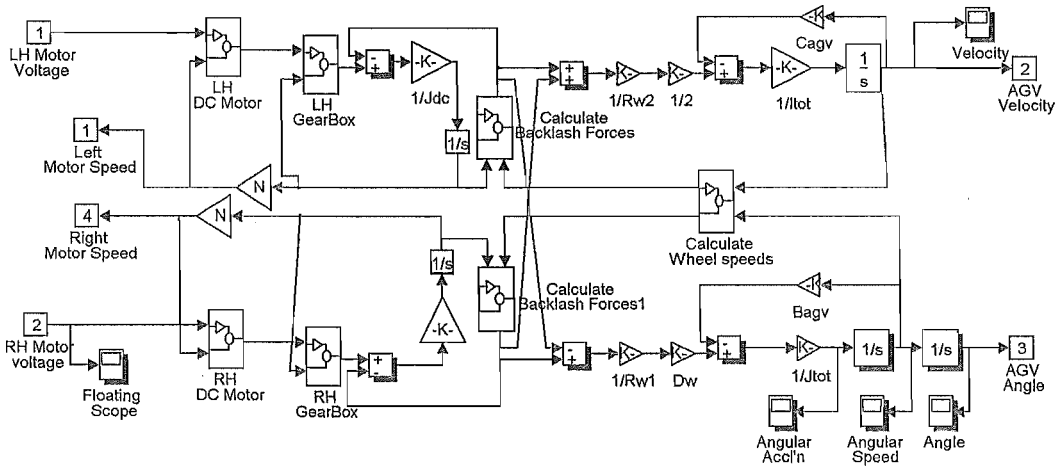
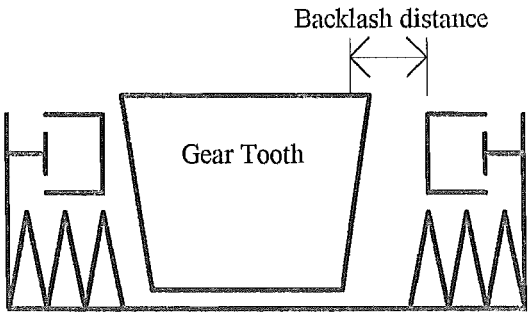


Fig. 2.17- Simulink screen: AGV kinematics complete with backlash.

The AGV is now treated as a uniform chassis under the influence of two point forces located at the wheels. Each of these forces is defined by the interaction the teeth of the mating gears in the gear train. The same forces that act to accelerate the AGV, act to slow down the DC motors.



Spring constant Ks
Viscous Damping constant Cs

Fig. 2.18- Diagram of the Gear Backlash Model.

The drive train is free to act under it's own inputs until the difference between the gear tooth position and the wheel position are great enough for the teeth to engage. The

teeth are treated as damped springs for ease of simulation. Modelling the teeth as springs is consistent with the engineering convention of treating gear teeth as cantilever beams under load.

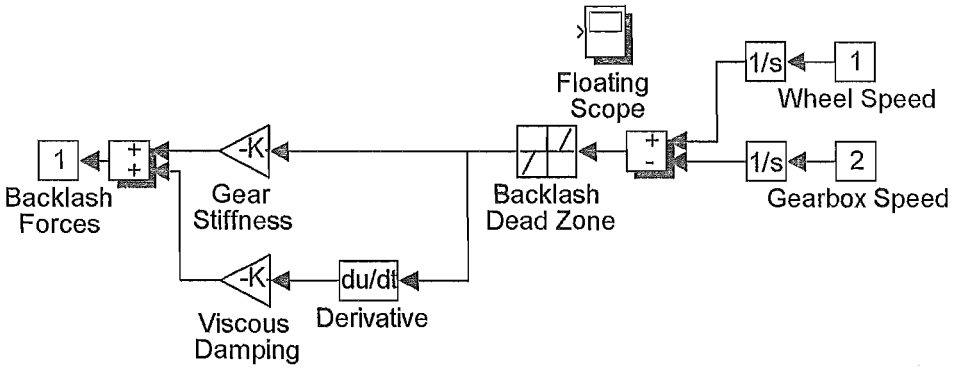


Fig. 2.19-Simulink Block diagram of the Gear Train Backlash Model.

The addition of the spring stiffness terms introduces new dynamic terms with natural frequencies many times greater than any others in the model. These new terms reduce the size of the minimum time step required to simulate the model to 0.005 seconds. As a means of attempting to reduce the effects of these new terms, viscous damping terms were added to minimise the reduction in time-step and speed up the simulation process, by increasing the damping ratio of the springs and reducing their natural frequency.

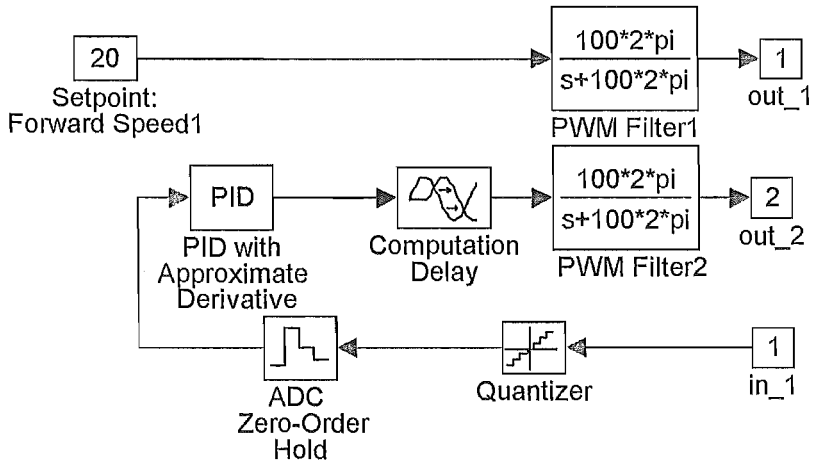


Fig. 2.20 - Complete 80c552 Microcontroller Block including; the low pass (-3dB @ 100Hz) filters on the PWM, timing delays and quantisation of the sensor signal.

3. AGV ELECTRICAL SYSTEMS

This section presents descriptions of the various electrical sub-systems on the UOC AGV MKII.

3.1 AGV ELECTRICAL SYSTEMS

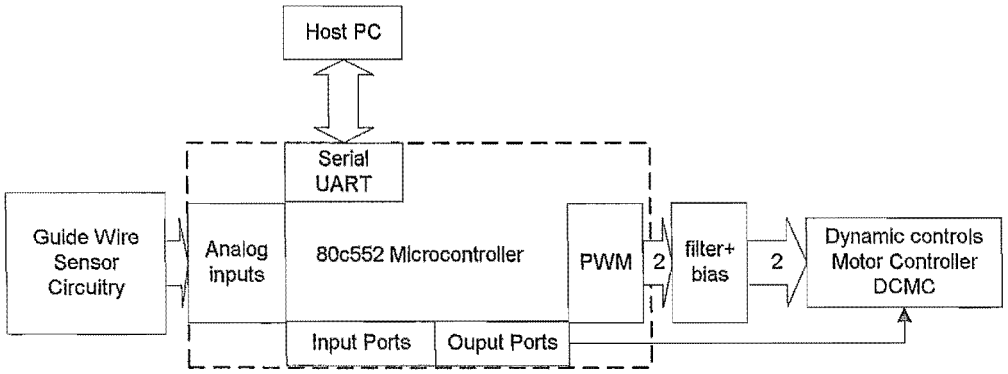


Fig. 3.1 The Microcontroller Control sub-systems

The AGV test system consists of five main sub-systems. On the AGV were the following 3 sub-systems:

- 80C552 Microcontroller Board
- 'Dynamic Controls' Motor Controller Interface
- Guide Wire Proximity Detection Circuit7

3.1.1 AGV 80C552 Microcontroller Board

The AGV microcontroller performed the input and output processing to enable the AGV to determine its state in the environment and to react by driving its wheels accordingly. The CPU was a Phillips 80C552 8-bit microcontroller which is a derivative of the Intel MCS51 family with additional on-chip peripherals, A/D conversion, Pulse Width Modulation (PWM) generators, digital timers and counters etc. These peripherals operate independently of the main CPU, and except for the actions of initiating, terminating or reading results from these peripherals, they do not impact on the microprocessor's performance.

The microcontroller was purchased complete as a development board for application testing and included an on board monitor program in EPROM for interfacing with a host computer via a serial communications port. This monitor program was later replaced by a more complicated 'C' debugger/monitor called the 'Lucifer Debugger' from HiTech Software.

An elementary user interface was added to the AGV. It consisted of an LCD display (20 character x 2 line array) and an array of push buttons. By means of the AGV user interface, control variables could be changed without the need to reconnect the AGV to it's support PC.

The microcontroller memory map consisted of 8k RAM and 8k ROM mapped into the same address space. The architecture of the 80c51 family of microcontrollers has separate program and data spaces. This feature is appropriate for commercial applications where the product's program is static, but it is not appropriate for code development. In order to program new code into RAM to be executed, the memory spaces had to be re-mapped to coincide by digital logic ('OR'ing PSEN and RD) in external circuitry. Thus programs could be read and written to high address data ("RAM") blocks which were also mapped as low address program ("ROM") blocks.

The microcontroller interfaces with a number of small peripheral circuits which were driven by the dedicated circuits on the microcontroller. The internal PWM circuit was used to generate an analog signal via a filter circuit. This analog voltage to the DCMC was used to indicate the Forward/Turn speeds.

3.1.2 Dynamic Controls Motor Controller (DCMC)

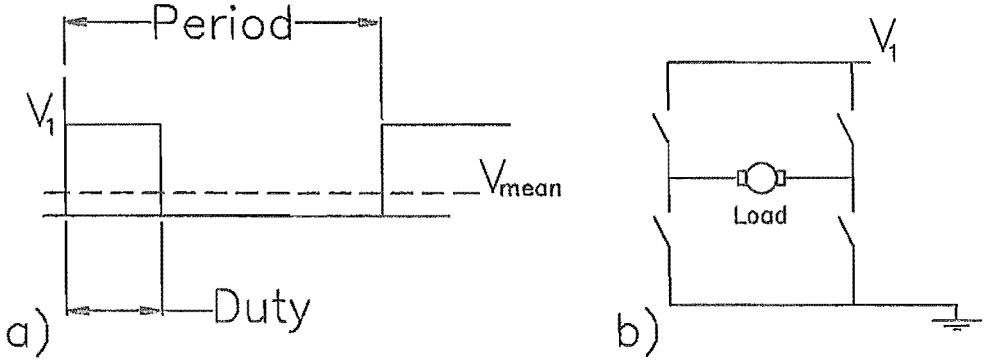


Fig. 3.2 (a) PWM duty cycle (b) H-bridge schematic

The DCMC is essentially a microprocessor controlled pair of H-bridge circuit drivers. It has circuitry to maintain and monitor the speed of the wheelchair motors to which it is usually attached. The exact circuitry of the DCMC is a commercial secret and is not available. The DCMC allows proportional speed control of the motors by means of PWM switching of the four arms of the H bridge. By switching quickly between 'on' and 'off' states, the mean voltage applied to the motor is proportional to the mean 'on' time of each PWM cycle.

The DCMC used a single input as an 'enable'. Failure to assert this input would engage a 'braking' mode to bring the AGV to a stop and attempt to hold the wheels stationary. From experience pushing the AGV from point to point between test runs, the resistance to motion was significantly greater when the DCMC was engaged than when it was off. This meant that the DCMC gave substantially better braking performance under the control of the DCMC than by simply removing the power (as was done to test the stopping distance for the bumpers).

As the DCMC was designed to control a wheelchair, it exhibited a number of operational 'quirks' which were necessarily accommodated for in the microcontroller program. If the DCMC received a non-zero demand signal in the first ½ second after triggering the enable, the controller would fault. This 'feature' was to prevent a wheelchair occupant from removing the park-brake and immediately going into motion, possibly injuring themselves in the process. As well, the microcontroller had limits to the demand signals it would accept. A combination of high forward speed demand, and high turn speed demand would cause the DCMC to indicate a fault and bring the motors to a stop. This prevents the wheelchair pilot being thrown out during a high speed turn. Thus speed must be reduced for sharp turns.

3.1.3 Guide Wire Proximity Circuits

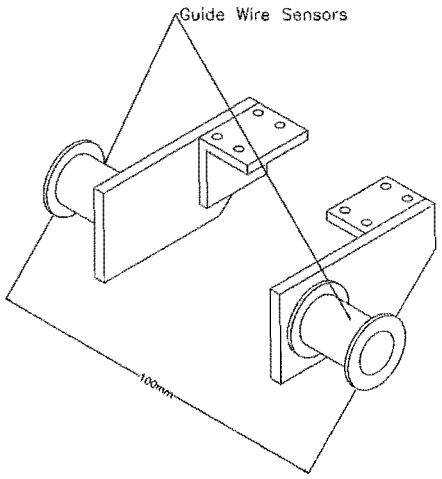


Fig. 3.3- Mechanical layout of the Guide Wire Sensors

The AGV used two tuned receivers to detect the proximity of the wire. The major components of the sensor system were a tuned LCR resonator circuit, an active full wave amplifier and an AM demodulating circuit.

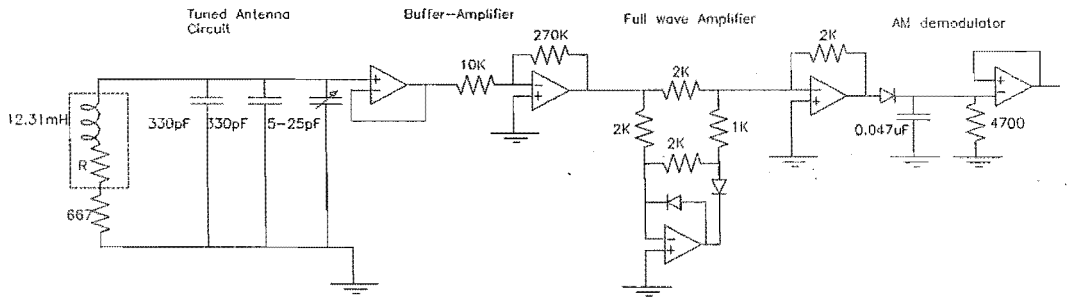


Fig. 3.4- Guide Wire Sensing Circuit

The Tuned Sensor Circuit was LCR circuit was designed to operate with a resonance at 45kHz, and a Q-factor of approximately 10%. The Q factor was chosen so that the AGV could use the guide wire as a communications link (not implemented to date) to the host PC by using frequency shift keying to modulate a digital signal onto the guidance waveform.

The sensors inductors were hand wound to a convenient size for handling and mounting to the AGV. Next component values were chosen to give the correct resonance, and the circuit was finally tuned to the guide wire frequency using the trim capacitors.

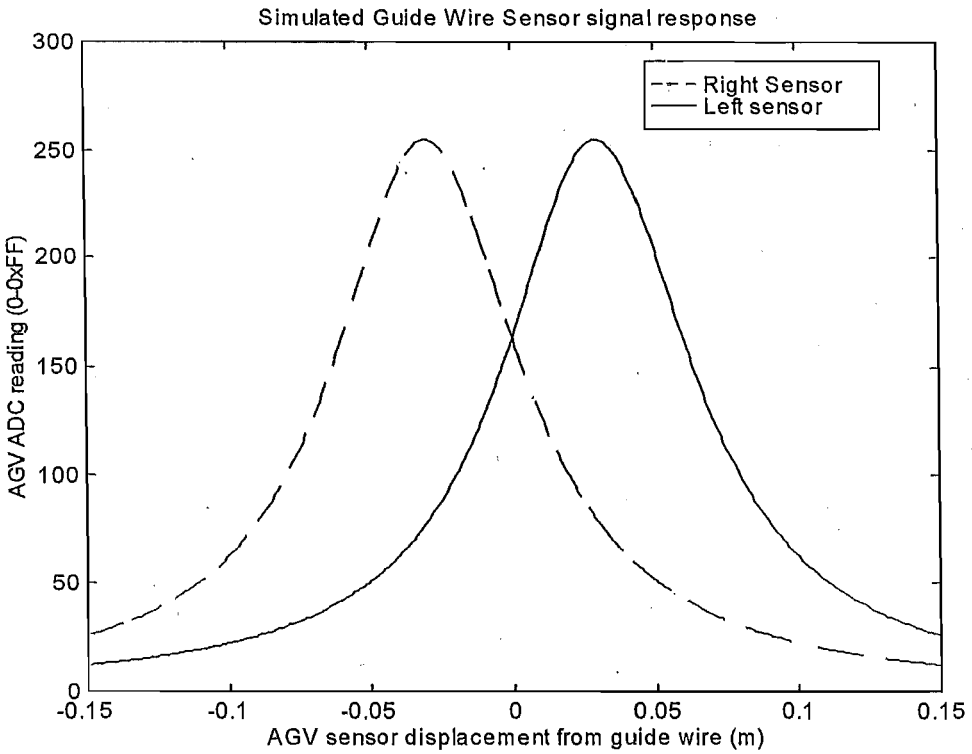


Fig. 3.5 - Guide Wire Sensor signal plot

The resonator signal was buffered and amplified using a high slew rate op-amp to bring the sensor signal to a level that the microcontroller could detect. The AM demodulating circuit removed most of the AC signal from the amplified signal, leaving only slight ripple in the signal to the microcontroller. The relatively low sampling frequency combined with the resolution, were such that the microcontroller wasn't able to detect any ripple present from the AM demodulation.

After tuning the receivers, the gain and offset of the AM demodulating circuit were calibrated to give equal values to the microcontroller (0xFA or 250 decimal) when the respective sensor was directly over the guide wire. In normal conditions the guide wire sensors attached to the AGV were approximately 38mm above the guide wire.

3.2 AGV SUPPORT ELECTRONICS

To run the AGV test system two support systems were required. The Guide Wire Signal Generator, provided the high frequency guidance signal for the AGV. While the Host Personal Computer (PC) provided programming facilities for the microcontroller and the data logging of experimental results.

3.2.1 Guide Wire Signal Generator

The guide wire was powered by a standard laboratory function generator. The function generator was connected to the guide wire loop circuit in series with a resistance, and set to produce a 45kHz sine wave.

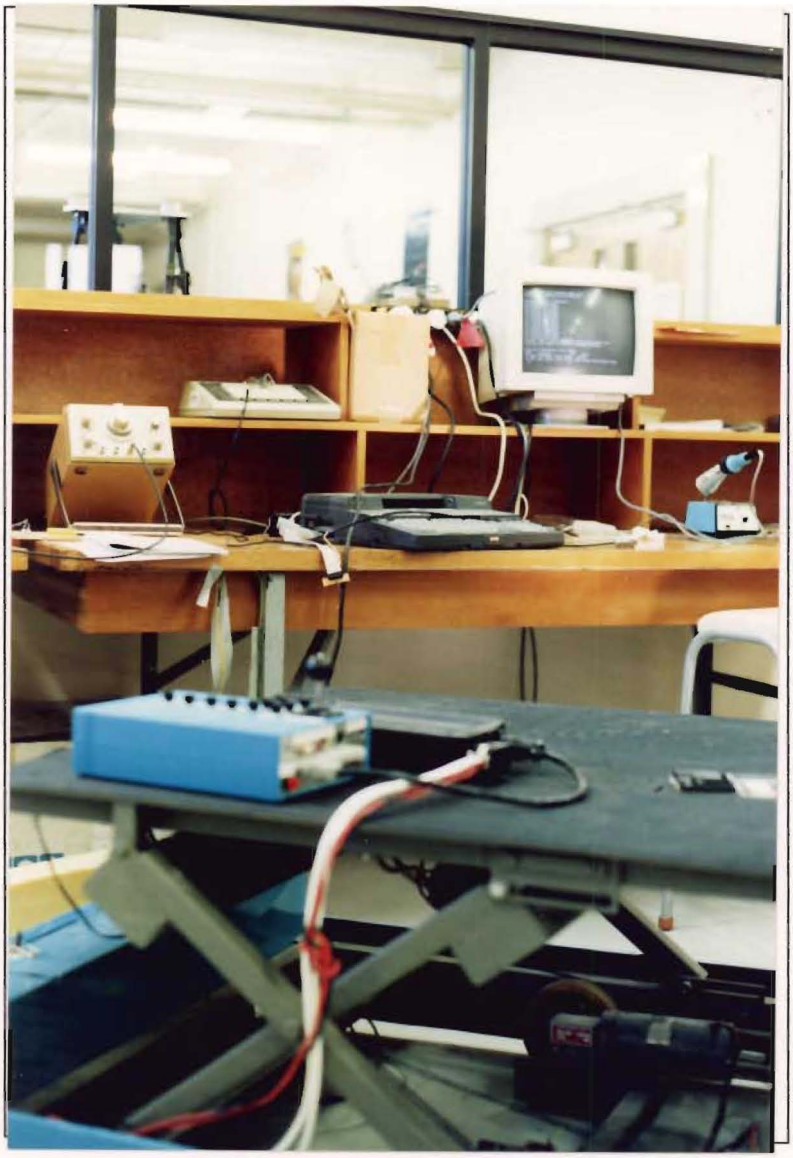


Fig. 3.6- Photo, The Experimental Rig

3.2.2 Host PC

An IBM-PC compatible computer was equipped with software and hardware, in order to program the microcontroller and to perform data recording for the system identification experiments.

The Universal Pulse Processor Card (UPP Card) was an ISA-bus card designed as a signal processing peripheral for the IBM-PC (c.f. G.R. Dunlop et al [21]). Past applications of the card have included; Fuzzy Logic control of an inverted pendulum, real time optimal control of a hydraulic cylinder (c.f. Steve Hampson [1]), position control of a prosthetic finger (c.f. Derek Ward [17]).

The UPP has digital pulse processing capabilities as well as A/D converters. The pulse processing circuitry is programmable, in that a dedicated controller on the card (Hitachi Part #HD64130) can be programmed to execute simple macro functions to registers on the chip. Programming libraries written by Andy Cree[1] and Julian Murphy[2] allowed an easy interface to the UPP for accessing the registers.

On the AGV project the UPP's digital pulse processing functions were primarily used to log speed and position information from quadrature encoders, and to generate timing pulses for other data logging operation. The encoders used were 2 channel 256 count per revolution quadrature encoders, giving 1024 position counts per revolution.

The UPP's analog inputs were used variously to measure analog voltages from the AGV, the demand signals to the DCMC, and the back EMF generated by the each DC motor. In each case, the voltages in the experiment were scaled to fit into the range of the A/D converters on the UPP by using simple OP-AMP circuits.

---

[1]Technician in charge of UOC Robotics and Applied Mechanics laboratories, PhD candidate

[2]Technician in charge of UOC Electronics and Computer workshop

4. AGV PROJECT SOFTWARE

During the course of the project, a number of computer programs were written to control the AGV and it's support systems. This section describes the software that was used to write these programs, and the programs that were written.

4.1 MICROCONTROLLER PROGRAMMING SOFTWARE

A number of languages were considered to program the microcontroller, including variants of 'Forth', 'Modula-2' and 'C'.

Assembly language was initially considered, but the lack of familiarity with the 80c552 made that option inefficient. The assemblers evaluated were mostly free-ware or share-ware, and mostly missing any facility for linking multiple files into a single block of machine code for execution. However over the duration of the project, a great many opportunities to become familiar with 80c51 assembly code presented themselves. These arose from the need to de-bug and evaluate code generated by the various compilers.

Dynamic Controls Ltd provided a copy of 'Mod51' a commercial 'Modula-2' package designed for the 80c51 family of microprocessors. This package was used in the first stages of software development, however it soon proved to be more difficult to use than was anticipated. The compiler generated extremely bulky code in certain circumstances, and the syntax of the 'Modula-2' language was restrictive compared to 'C'. The author was initially unfamiliar with 'Modula-2', and sought opinions of the 'Mod-51' compiler from more experienced 'Modula-2' programmers. The consensus view was that this implementation was incomplete in terms of it's functionality and lack of adherence to the de-facto industry standards. Henceforth, in concert with another project in progress within the department, a 'C' development suite was purchased.

The 'HiTech -C' development suite consisted of the usual bevy of 'C' tool including pre-processor, linker, and an Integrated Development Environment (IDE). As well, it included an extension to the IDE in the form of a program to download software to a target system, and debug the users 'C' source code. This program, 'The Lucifer

Debugger' (LUC51.EXE), used the serial port of the PC to communicate to a microcontroller running the 'Lucifer target'. This enabled the programmer to step through the 'C' program running on the microcontroller, and see the corresponding section of code on the PC screen. The 'Lucifer Target' was a monitor program stored in EPROM on the target microcontroller. On the AGV the 'Lucifer Target' program replaced the standard development EPROM supplied by the development board manufacturer (Phillips).

When the microcontroller booted, the program would test the status of one of the input lines. Depending on the state of this line, it would either execute the 'Lucifer Target' or a program stored in user RAM. A section of code would be loaded into RAM on the Microcontroller, debugged using the 'Lucifer' and then the microcontroller would be re-booted after switching the 'User/Monitor' switch to 'User'. Upon a hard RESET the microcontroller would execute the user code in RAM, ignoring the 'Lucifer' code.

4.2 Host PC Support Software

The software to perform data-logging of the AGV for the System Identification experiments, was written using 'Borland-C++ V1.0' and the UPP libraries written by Andy Cree and Julian Murphy (See Section 3.2.2).

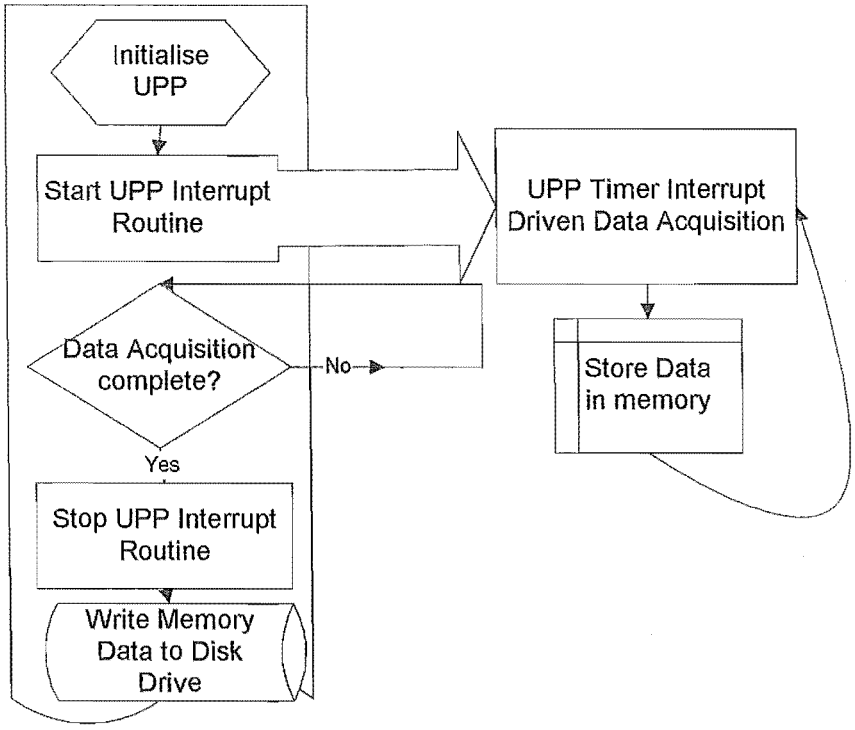


Fig. 4.1- Block Diagram, software functions in IDSYS.C

The programs written to log data for the system identification experiments followed a common design strategy. The UPP was set to generate timed interrupts from an internal register. Upon generation of an interrupt, the host PC would execute an interrupt handling routine that would load the experimental data from the relevant registers in the UPP and then store them in a buffer assigned to the task. At the end of an experimental run, the UPP program was terminated and data in the memory buffer written to a disk file for future analysis.

For example, the program IDSYS.EXE was written in 'C' code to record the demand signal to the DCMC and the corresponding speed output of the DC motor. This program used the UPP to generate interrupts at a maximum speed of one interrupt per millisecond. Each time the interrupt was serviced it would initiate an A/D conversion to record each of the DCMC demand signals and record the value of the Up-Down counters that recorded the DC motor positions from the quadrature encoders. At the end of the timing run (typically 10 seconds), the file of the results was recorded on the hard disk, along with a comment and a time and date record of the experiment.

4.3 AGV MICROCONTROLLER SOFTWARE

The programs to control the AGV were written using 'Hi-Tech C'™, compiled, linked and down-loaded to the microcontroller from within the 'Hi-Tech' IDE. This (nearly) seamless path meant that relatively quick code development was possible.

4.3.1 Embedded Controller Software

The core of the microcontroller software consisted of an interface between the AGV's inputs and outputs, and the control algorithm.

The basis of embedded control is the use of a constant and known time-step to execute a timed control algorithm. This allows discrete time control algorithms (eg. difference equations) to be formulated off-line (on the Host PC for example) and ported to the microcontroller. The 80C552 microcontroller's on-board peripherals are designed for embedded control applications.

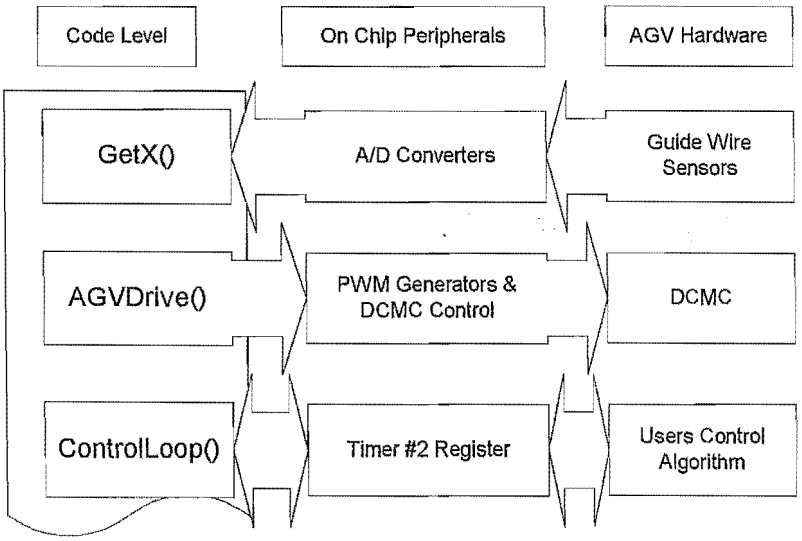


Fig. 4.2 - Diagram of AGV Control Software Interface

The architecture of the control software for the AGV follows a form common to the code used for the data-logging with the UPP card on the Host PC. The AGV uses one of the on-board timer circuits to generate an interrupt at known intervals. This interrupt executes the control algorithm and allows the interrupt to conclude and return to other tasks in time for the next control loop ie. the regular interrupt tasks are always completed before the next interrupt occurs. The programs are:

AGVLOOP.C : Uses on-board timer (TOC2) to generate timed interrupts to a service routine. The service routine is vectored to the address of the users control code, as defined in the arguments passed to the 'StartControlLoop()' function.

AGVAD.C : Retrieves the values of the ADC from the guide wire sensor inputs in round robin fashion and stores the resultant values in the appropriate locations.

AGVPOS.C : Calculates the position of the AGV's Sensor from the sensor inputs incident at the ADC.

AGVCONT.C : This module calculates the control signal to pass to AGVDRIVE() from the position of the AGV calculated in AGVPOS.C.

AGVDRIVE.C : Performs the task of interfacing the users control code with the AGV drive algorithms. The routines control and manage the drive status (Enabled or Disabled) and the speed demand sent to the DCMC.

In general the software to control the DCMC via the microcontroller was developed by trial and error in the laboratory. Simply put, an experimental methodology was used to trial timing and demand limits to the DCMC in order to get the maximum performance from the AGV without the DCMC faulting. Trial speed demands were sent to the DCMC (with the wheels of the AGV safely off the ground) and limiting demands were established. These limits were hard coded into the control software as constants. Thus operating limits were established empirically which avoided generating DCMC faults.

4.3.2 Utility and User Interface Software

In the main loop, the program sent status information to the LCD panel and monitored the buttons for input. It also monitored the internal state of the AGV for events such as collisions. The programs were:

AGVSER.C : Interfaces the user program with the on-chip UART. This section of code provides elementary serial communications functions between the Host PC and the microcontroller. This module provides functions to send bits, bytes, and words as decimal or hexadecimal.

AGVINT.C : This code module sets up the AGVs bumper safety interrupt. Upon triggering by the AGV's bumper, the interrupt routine sets the DCMC speed demand signals to zero (50% PWM) and disables the drive.

The bumper interrupt operates on the XIRQ line to the microcontroller, which is a Non-Maskable Interrupt (or NMI). This executes a jump to a routine which removes the 'DriveEnabled' bit disabling the DCMC. The 'DriveEnabled' bit is a flag used to indicate the status of the DCMC to the rest of the AGV software. It should only be cleared by the bumper interrupt routine and written to by a Master Reset. This is simply a software safety interlock. As such it should not be trusted in any other than a controlled laboratory environment, as any errant code may accidentally overwrite the wrong bits and send the AGV careering out of control. For commercial applications, a hardware/hard-wired safety relay would remove power to the DCMC until the AGV is reset by an authorised operator.

AGVVARS.C : This module implements the variable adjustment program via the keyboard and LCD screen on the AGV. It enables the user to change variables within the program provided they are listed in the program at compile time.

The User interface routines provided the means for the operator to vary certain operating parameters within the AGV. Mostly, this consisted of maintaining the control value coefficients. The user interface was written so that the operator could modify the numerator and denominator separately, so as to obtain practically any coefficient that can be represented fractionally.

The 80C552 microcontroller can only perform 16 bit maths (fixed precision). The simplest method for implementing fixed precision maths without increasing the computational load on the CPU significantly by using a floating point library, is to use a numerator and denominator value to represent a single fractional value. In this way control values are calculated by first multiplying by the numerator, and then dividing by the denominator.

4.3.3 The P-D Controller PLAYTIME.C

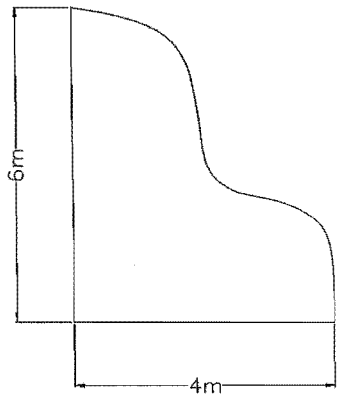


Fig. 4.3- Guide Wire circuit for the AGV running PLAYTIME.C

After testing the individual sections of code, the first implementation of the complete AGV control system was a demonstration of the AGV for a departmental Open Day.

On this occasion the AGV test rig was set up to follow a loop wire on the ground, and exhibit simple 'searching' behaviour if it lost the wire. The AGV drives forward at low speed until it drives off the guide wire. The AGV stops and spins until the Guide Wire signal strength is greater than an arbitrary value. The AGV re-engages the wire-following P-D control and continues until it loses the wire again.

PLAYTIME.C used the AGV software routines to follow the wire using a simple Proportional and Derivative (P-D) control law. It was this control law which was used to control the AGV for the laboratory test runs.

Quoting the 'C' code directly;

```
void InitControl(void)
{
        Kpt.num = -1;
        Kpt.den = 20;

        Kdt.num = 1;
        Kdt.den = 8;

        Kit.num = 0;
        Kit.den = 1;

};
```

The proportional and derivative gains are set to -1/20 and 1/8 respectively.

```
void GetU(void)
{
        /* Rotate the X values along one in the X array */
        X[3] = X[2];
        X[2] = X[1];
        X[1] = X[0];
```

```
        X[0] = (Xpos);

        /* Rotate the U values along one in the U array */
        U[3] = U[2];
        U[2] = U[1];
        U[1] = U[0];
        U[0] = U0;

        U0.t = Kpt.num*(R.t - X[1]) / Kpt.den + Kdt.num*(X[0]-X[1])/Kdt.den;
        U0.f = R.f;

};
```

The previous control values are stored in a series of buffers. Before a new control value is calculated, the old values are rotated forward in the arrays. This implementation is relatively CPU intensive given that block move instructions such as those found on the Z80 are not available on the MCS51 microcontrollers. Also the assembly code generated by the 'C' compiler for array manipulations is not compact.

A better implementation of this would be to use a circular buffer and use a single pointer to the most recent value. If the buffer is 2^N long (where N is an integer) and starts at an address H where the lower N bits are zero, then an effective circular buffer can be generated by always 'OR'ing any buffer address with H after setting the top bits to zero. Thus if the address is incremented past the end of the buffer, the AND 00..0011..1 (where there are N 1's) followed by OR H (where there are N 0's) always puts the address pointer back to the start of the linear address block. Hence the address scheme acts as a circular buffer. Note, if H is all 1's followed by N 0's (ie. the top 'page' of memory) then only the OR operation is required.

From these buffered control and sensor values, the P-D control algorithm calculates an approximate derivative and a new control value.

5. RESULTS OF SYSTEM IDENTIFICATION EXPERIMENTS:

This section presents and examines the results of the experiments performed to analyse the dynamics of the DCMC and the DC motors. The data was processed on Matlab, and plots were generated comparing the simulated performance of the parameter estimated models and the actual experimental data. To demonstrate these results a typical series of plots is presented here, and a full summary of the data is given in the appendices.

5.1 Analysing The Electric Motors And Drive Train Recording

To measure the values of the AGV's physical properties, a number of short tests were performed. These were described in Section 2.4.1. These results were used to calculate the coefficient of the Back EMF, and also the rotational inertial of the rotor for the DC motors.

Section 2.4.2 described the experiment to determine the Back EMF constant, and the angular inertia of the DC motor. Examination of the plot of the angular velocity-vs-time graph in fig.5.1 shows that the DC motor rotor accelerates until the weight hits the ground and then slows to a stop. The velocity of the DC motor under deceleration shows an almost constant deceleration ie. none of the characteristics of a body under viscous damping. The major braking force would seem to be a constant friction force due to the bearings and the DC armature brushes.

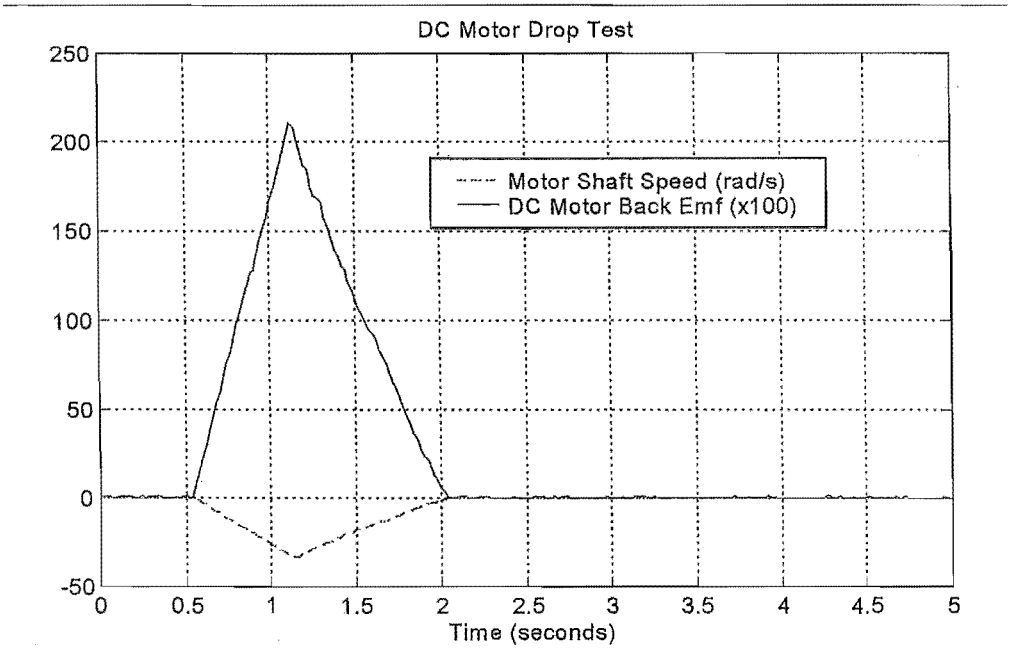


Fig. 5.1- Results of the Falling Weight test.

Assuming that the viscous damping in the DC motor was negligible in this test, values for the DC motor braking force and angular inertia were calculated. From the measurement of the Back EMF constant Kv and the rotor inertia were calculated:

$$K_V = 0.063 \text{ V rad}^{-1}\text{s} \tag{32}$$

$$J_{DC} = 2.2 \times 10^{-4} \text{kgm}^2 \tag{33}$$

5.2 System Identification Analysis Of The DCMC Controlling A DC Motor

The following sections are used to examine the possible DCMC models which were suggested in section 2.3. The equations of motion and the associated simulation block diagrams are also developed in each section.

In each case a model is proposed to represent the dynamics of the DCMC. State space matrices are derived from the equations of motion and the unknown parameters are shown in the parameter estimate matrices. Using the PEM.M function from the System Identification Toolbox, the unknown parameters are estimated using a least squares error minimisation method. This attempts to minimise the error between the output of a simulated time series produced by presenting the data readings of the

system inputs to the estimated parameter model, with the measured outputs of the system.

The parameter estimation methods employ discrete time steps using the data-logging interval as the time-step to compare the estimated response and the measured response. After the parameter estimation is complete, either by finding stable solutions to the parameters, or by exceeding a given number of iterations, the resulting estimated model is converted back to a continuous time model for presentation. In the estimation process the estimate model is given a confidence interval. The confidence interval (of one standard deviation) represents the range of parameters which would represent alternative values for the calculated model parameters.

In the case of the DCMC, estimates were made of the unknown parameters, and used as first guesses for the estimation process. As the PEM function uses a gradient search method to minimise the parameter error, it sometimes converges to local minima as a result of poorly chosen initial conditions. Simulation of such estimated models might show close correlation of one simulated output and inadequate tracking in another. By re-substituting apparently successful values into the initial conditions at the beginning of subsequent runs, further initial solutions were found that converged more quickly to valid parameter estimates.

If a good solution was found for a parameter then by taking advantage of the symmetry of the model (in this case), the value could be substituted into another corresponding parameter. Some of the parameters were expected to be fairly similar (as the DCMC was designed to run two similar motors). For example in the next section, if in Case 2-I the parameter P_{B11}^2 provided a valid solution to K_{11} and the value P_{B12}^2 couldn't be solved adequately (for K_{12}) then the next iteration of P_{B12}^2 would be initialised with the value from P_{B11}^2.

The results shown in the sections are from the PEM function using the data from the data file 'RW21.DAT'.

5.2.1 Case 1-I Modelling the DCMC as a simple gain.

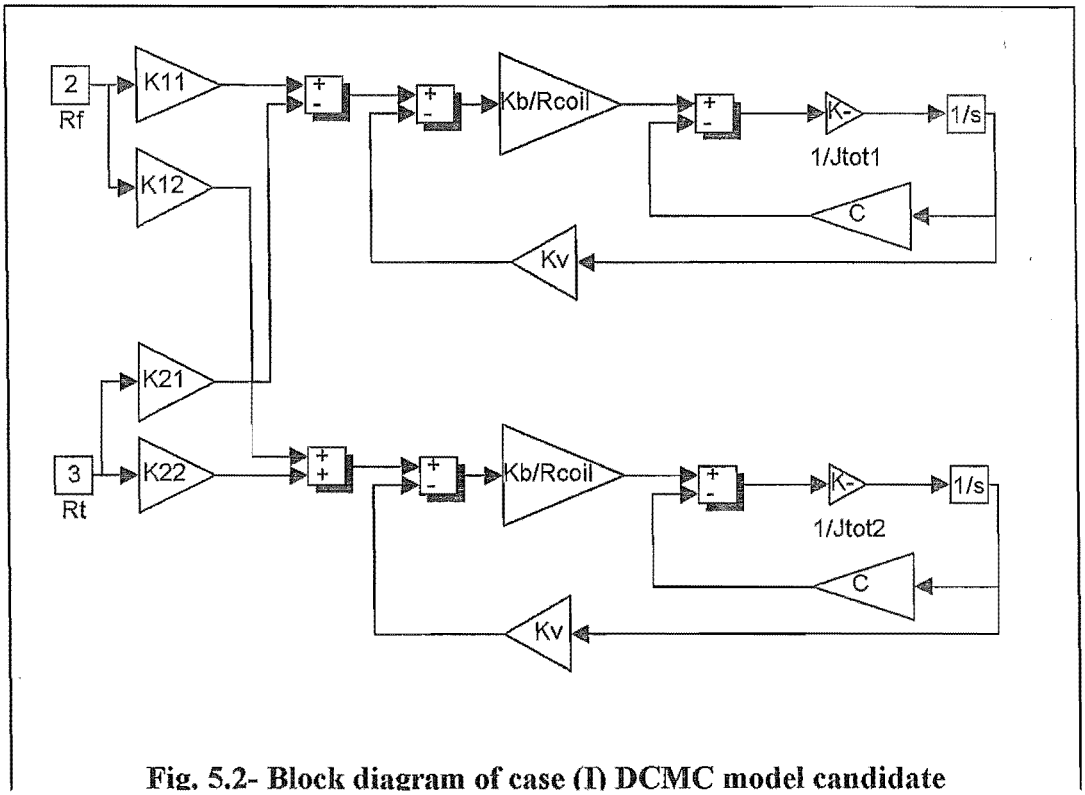


Fig. 5.2- Block diagram of case (I) DCMC model candidate

The equation of motion for a single DC motor is (neglecting rotor inductance);

$$s[\Omega] = \left[-\frac{1}{J_{eff}} \left(C + \frac{K_B K_V}{R} \right) \right] [\Omega] + [K] R \tag{34}$$

Combining the simple single motor case as two DC motors connected to the DCMC, the state matrices become;

$$s \begin{bmatrix} \Omega_l \\ \Omega_r \end{bmatrix} = \begin{bmatrix} -\frac{1}{J_{eff}} \left(C + \frac{K_B K_V}{R} \right) & 0 \\ 0 & -\frac{1}{J_{eff}} \left(C + \frac{K_B K_V}{R} \right) \end{bmatrix} \begin{bmatrix} \Omega_l \\ \Omega_r \end{bmatrix} + \begin{bmatrix} K_{11} & K_{21} \\ K_{12} & K_{22} \end{bmatrix} \begin{bmatrix} R_t \\ R_f \end{bmatrix} \tag{35}$$

Where the system is of the form;

$$\dot{x} = \mathbf{A}x + \mathbf{B}u \tag{36}$$

$$y = \mathbf{C}x + \mathbf{D}u \tag{37}$$

the state space matrices are;

$$A = \begin{bmatrix} -\frac{1}{J_{eff}}\left(C + \frac{K_B K_V}{R}\right) & 0 \\ 0 & -\frac{1}{J_{eff}}\left(C + \frac{K_B K_V}{R}\right) \end{bmatrix} \quad (38)$$

$$B = \begin{bmatrix} K_{11} & K_{21} \\ K_{12} & K_{22} \end{bmatrix} \quad (39)$$

$$C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (40)$$

$$D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad (41)$$

Using this matrix as the form of the model, the System Identification Parameter model is produced as follows;

$$A = \begin{bmatrix} P^1_{A11} & 0 \\ 0 & P^1_{A22} \end{bmatrix} \quad (42)$$

$$B = \begin{bmatrix} P^1_{B11} & P^1_{B12} \\ P^1_{B21} & P^1_{B22} \end{bmatrix} \quad (43)$$

As the matrices C and D have no unknowns, they remain unchanged.

In the following frame shows the results of the Matlab PEMM.M function. The terms of the matrices a, b and c are the model representations. Where a term is imaginary, the imaginary part is the first standard deviation for the value of parameter that fits the model.

```
This matrix was created by the command PEM      on 2/22 1997 at 17:17
Loss fcn: 4.45e+005   Akaike's FPE: 4.54e+005 Continuous time model
estimated using sampling interval 0.025
The state space matrices with their standard deviations given as imaginary
parts are
a =
  -1.5447 + 0.0533i         0
        0             -1.7278 + 0.0315i
b =
  -4.3438 + 0.1248i  -3.1130 + 0.1074i
  -3.9616 + 0.0641i   2.8626 + 0.0498i
c =
     1     0
     0     1
```

For example the first term of the b matrix;

$$b_{1,1} = -1.5447 + 0.0533i \tag{44}$$

$$b_{1,1} = P^1_{B11} + \lambda^1_{B11} i \tag{45}$$

gives an estimated parameter of

$$P^1_{B11} = -1.5447 \tag{46}$$

with a first standard deviation of

$$\lambda^1_{B11} = 0.0533 \tag{47}$$

representing the set of values that this parameter could take. The lower the standard deviation, the greater the likelihood that the estimated parameter is representative of the system parameter for the given training data. The standard deviation can be greater for a given noisy set of training data, than if the training data is filtered to eliminate measurement noise or known unwanted disturbances.

5.2.2 Case 2-I Modelling the DCMC as a simple feedback controller.

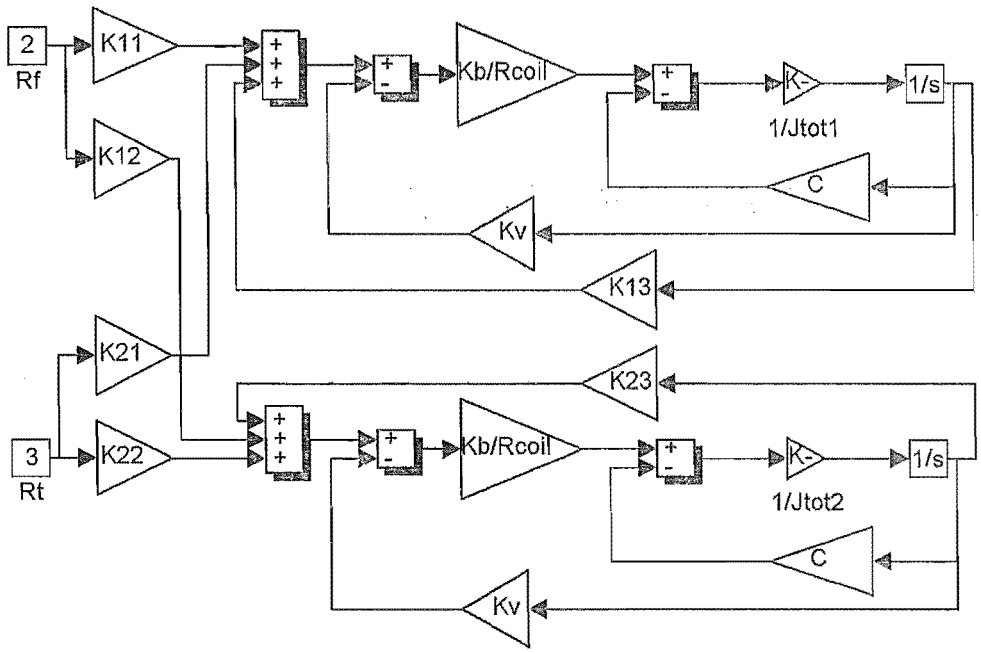


Fig. 5.3- Block diagram of case 2-I DCMC model candidate, Simple Feedback Controller

As for the first case, we examine the equations of motion of the AGV and consider the block diagram.

The using the block diagram to find the state matrices;

$$s\begin{bmatrix}\Omega_l\\ \Omega_r\end{bmatrix}=\begin{bmatrix}-\frac{1}{J_{eff}}\left(C+\frac{K_B\left(K_V+K_{13}\right)}{R}\right) & 0\\ 0 & -\frac{1}{J_{eff}}\left(C+\frac{K_B\left(K_V+K_{23}\right)}{R}\right)\end{bmatrix}\begin{bmatrix}\Omega_l\\ \Omega_r\end{bmatrix}+\begin{bmatrix}K_{11} & K_{21}\\ K_{12} & K_{22}\end{bmatrix}\begin{bmatrix}R_t\\ R_f\end{bmatrix} \quad (48)$$

Similarly, a parameter estimate matrices for this model become;

$$A=\begin{bmatrix}P^2_{A11} & 0\\ 0 & P^2_{A22}\end{bmatrix} \quad (49)$$

$$B=\begin{bmatrix}P^2_{B11} & P^2_{b12}\\ P^2_{b21} & P^2_{b22}\end{bmatrix} \quad (50)$$

Since C and D have no unknown parameters, they remain unchanged. Examining the terms of the A matrices in cases 1 and 2-1 for the situation where $K_{13}=0$, then

$$-\frac{1}{J_{eff}}\left(C+\frac{K_BK_V}{R}\right)=-\frac{1}{J_{eff}}\left(C+\frac{K_B\left(K_V+K_{13}\right)}{R}\right) \quad (51)$$

$$P^1_{A11}=P^2_{A11} \quad (52)$$

similarly

$$P^1_{A11}=P^2_{A11} \quad (53)$$

when $K_{23}=0$. Thus case 1 is a special case of 2-I where the feedback gain is zero.

In the situation where the system dynamics and the system coefficients are unknown, many of the coefficients cannot be calculated explicitly. In the case of the AGV, the viscous damping constant C is unknown and not easily calculable. From the results of the weight drop test, it was determined that the motor was mainly subject to coulomb friction, which would be much more difficult to determine.

```
This matrix was created by the command PEM      on 2/22 1997 at 16:40
Loss fcn: 4.45e+005   Akaike`s FPE: 4.54e+005 Continuous time model
estimated using sampling interval 0.025
The state space matrices with their standard deviations given as imaginary
parts are

a =

  -1.5447 + 0.0533i         0
        0             -1.7278 + 0.0315i


b =

  -4.3438 + 0.1248i  -3.1130 + 0.1074i
  -3.9616 + 0.0641i   2.8626 + 0.0498i


c =

     1     0
     0     1
```

Numerically, the results for case 1 and case 2-I are identical, the difference primarily in the interpretation of the results.

5.2.3 Case 2-II Modelling the DCMC as a P-I feedback controller.

The final model proposed for the DCMC was a PI controller. This model includes two extra states which monitor the wheel speed error, and adding an integral term. The model is as follows:

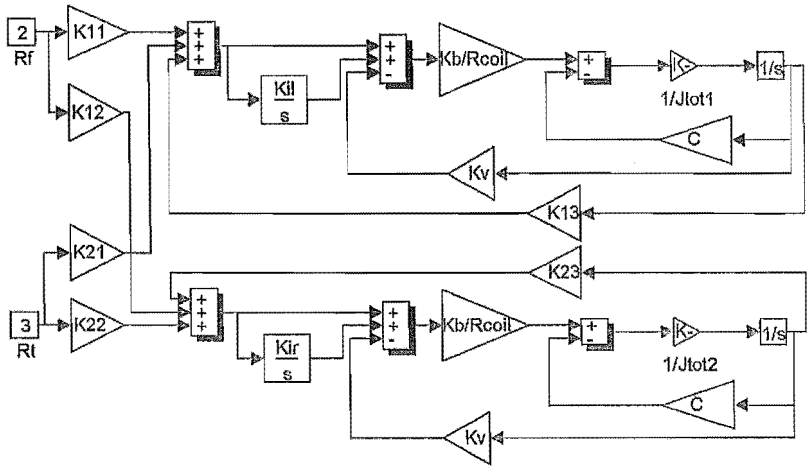


Fig. 5.4- Block diagram of case 2-II DCMC model candidate, PI Feedback Controller

By using the block diagram to find the state matrices;

$$s\begin{bmatrix}\Omega_l\\ \Omega_r\\ \varepsilon_l\\ \varepsilon_r\end{bmatrix}=\begin{bmatrix}-\frac{1}{J_{eff}}\left(C+\frac{K_B\left(K_V+K_{13}\right)}{R}\right) & 0 & 1 & 0\\ 0 & -\frac{1}{J_{eff}}\left(C+\frac{K_B\left(K_V+K_{23}\right)}{R}\right) & 0 & 1\\ -K_{13}K_{Il} & 0 & 0 & 0\\ 0 & -K_{23}K_{Ir} & 0 & 0\end{bmatrix}\begin{bmatrix}\Omega_l\\ \Omega_r\\ \varepsilon_l\\ \varepsilon_r\end{bmatrix}+\frac{K_B}{J_{eff}R}\begin{bmatrix}K_{11} & K_{21}\\ K_{12} & K_{22}\\ K_{11}K_{Il} & K_{21}K_{Il}\\ K_{12}K_{Ir} & K_{22}K_{Ir}\end{bmatrix}\begin{bmatrix}R_t\\ R_f\end{bmatrix} \tag{54}$$

This gives parameter estimate matrices of;

$$A=\begin{bmatrix}P^3_{A11} & 0 & 1 & 0\\ 0 & P^3_{A22} & 0 & 1\\ P^3_{A31} & 0 & 0 & 0\\ 0 & P^3_{A42} & 0 & 0\end{bmatrix} \tag{55}$$

$$B=\begin{bmatrix}P^3_{B11} & P^3_{B12}\\ P^3_{B21} & P^3_{B22}\\ P^3_{B31} & P^3_{B32}\\ P^3_{B41} & P^3_{B42}\end{bmatrix} \tag{56}$$

With a total of 12 unknowns, finding a valid solution with this model proved more difficult than with the other models. Repeatedly, the initial conditions for the parameter estimates lead to unstable solutions, or solutions that would only converge for one output.

```
This matrix was created by the command PEM      on 2/22 1997 at 16:45
Loss fcn: 3.094e+005   Akaike`s FPE: 3.22e+005 Continuous time model
estimated using sampling interval 0.025
The state space matrices with their standard deviations given as imaginary
parts are

a =

  1.0e+002 *

  -0.7415 + 1.2366i         0              0.0100                  0
        0             -0.2256 + 0.0410i         0              0.0100
  -1.1769 + 1.9936i         0                   0                  0
        0             -0.4237 + 0.0799i         0                  0


b =

  1.0e+002 *

  -0.0317 + 0.0286i   0.0387 + 0.1167i
  -0.0099 + 0.0059i   0.0002 + 0.0054i
  -3.0932 + 5.2568i  -2.3636 + 4.0009i
  -0.9311 + 0.1769i   0.6603 + 0.1254i


c =

     1     0     0     0
     0     1     0     0
```

In other estimation runs not documented here for brevity, the number of parameters to estimate was reduced by proposing that the PI model is a simple extension of case II. Results from the previous estimated models were substituted into the appropriate terms in the new model, so that the parameter estimator was only solving for the new parameters containing the gains K_{13}, K_{23}, K_{il} and K_{ir}. The results of this technique were no more successful and yielded results which were nearly identical to those of Case-I, as such these results are not illustrated.

5.2.4 Discussion of the system identification models

The primary purpose of the system identification process was to determine whether the DCMC had any higher-order control actions that could be modelled in the AGV simulation.

Examining the general form of the graphs, it is apparent that the calculated models approach the performance of the DCMC system. What is of particular interest is the behaviour of the models at the points where the output velocity changes near its extremes. The simulated model performance differs from the actual system at extremes of velocity, but matches it well in the interim.

Generally, the differences in performance may be attributed to two major factors:

- Mechanical effects on the DC motor. In all the test runs, the drive wheels and drive train were attached. As mentioned previously, significant backlash was present in the drive train. It is possible that the backlash is becoming apparent in the load presented to the DC motor thus creating a varying inertia load as the gears engage and disengage with the change in relative velocity. Also the DC motor was subject to Coulomb friction rather than viscous friction. These effects were not modelled in parameter estimation of the DCMC model.

- The DCMC in known to perform some artificial intelligence with respect to providing turning velocity at high velocity. Under normal circumstances, the order to turn given by the wheel chair's pilot is deemed to have priority over the average forward speed. This means that for a given turn input at near maximum forward speed, the DCMC will reduce the speed of the inside wheel by what it would have increased the outside wheel, plus it's own speed reduction. The parameter estimation method used assumes that the parameters being calculated are time-invariant. This may not be the case at the limits of velocity and turn demands.

In both cases, the simulated result has been given initial conditions of zero. In most cases, the experimental data was gained with the DC motors initially at speed, and as such the initial conditions of the experimental data were non-zero. The section of the graph in the first two to three seconds show the simulation differing from the experimental results significantly. After this period, the simulated results appear to follow the experimental results more closely.

Because cases (1) and (2-I), use the same parameter matrices (ie. the same number of unknown parameters in the same locations) the results of the numerical parameter estimation calculations were identical. As the numerical results of the estimation are identical, the performance of the two models with the simulated output is the same and thus it is impossible to determine which of the models is correct. As C cannot be accurately determined by the nature of the friction, the terms;

$$\left(C+\frac{K_B K_V}{R}\right) \textbf{ and } \left(C+\frac{K_B(K_V+K_{13})}{R}\right), \tag{57}$$

were considered to be treated as part of the damping term for the DC motor. This left the last decision as to which order of model best represented the DCMC.

Examining fig. 5.5 and fig. 5.6, in which the graphs represent the performance of the parameter estimated models compared with the actual experimental data, it is apparent that the proposed models are good approximations to the actual systems.

The System Identification tools used give a measure of the parameter fit to the experimental data. This fit is expressed as the 1st standard deviation of the value of the parameter, which gives a measure of confidence to the experimental data. For the PI controller the 1st standard deviation was typically 12-15% of the value of the parameter. For the simple gain, the fit was typically 1.2-1.7%. The additional fitting of the PI control model may simply have been a result of increasing the order of the model and over-fitting with noisy experimental data. The increase in the standard deviation was taken to be a reduction in the confidence of the accuracy of the model.

For the purposes of simulating the DCMC, the main consideration was the performance of the model versus its complexity. The addition of two new states into the AGV model would complicate the model further with parameters of dubious accuracy. It would also slow the simulation process with little perceivable improvement in the model results. Thus the DCMC was not treated as a PI controller in the AGV simulations.

Fig. 5.5 rw21.dat, TRYSS2.M, DCMC model 2-I

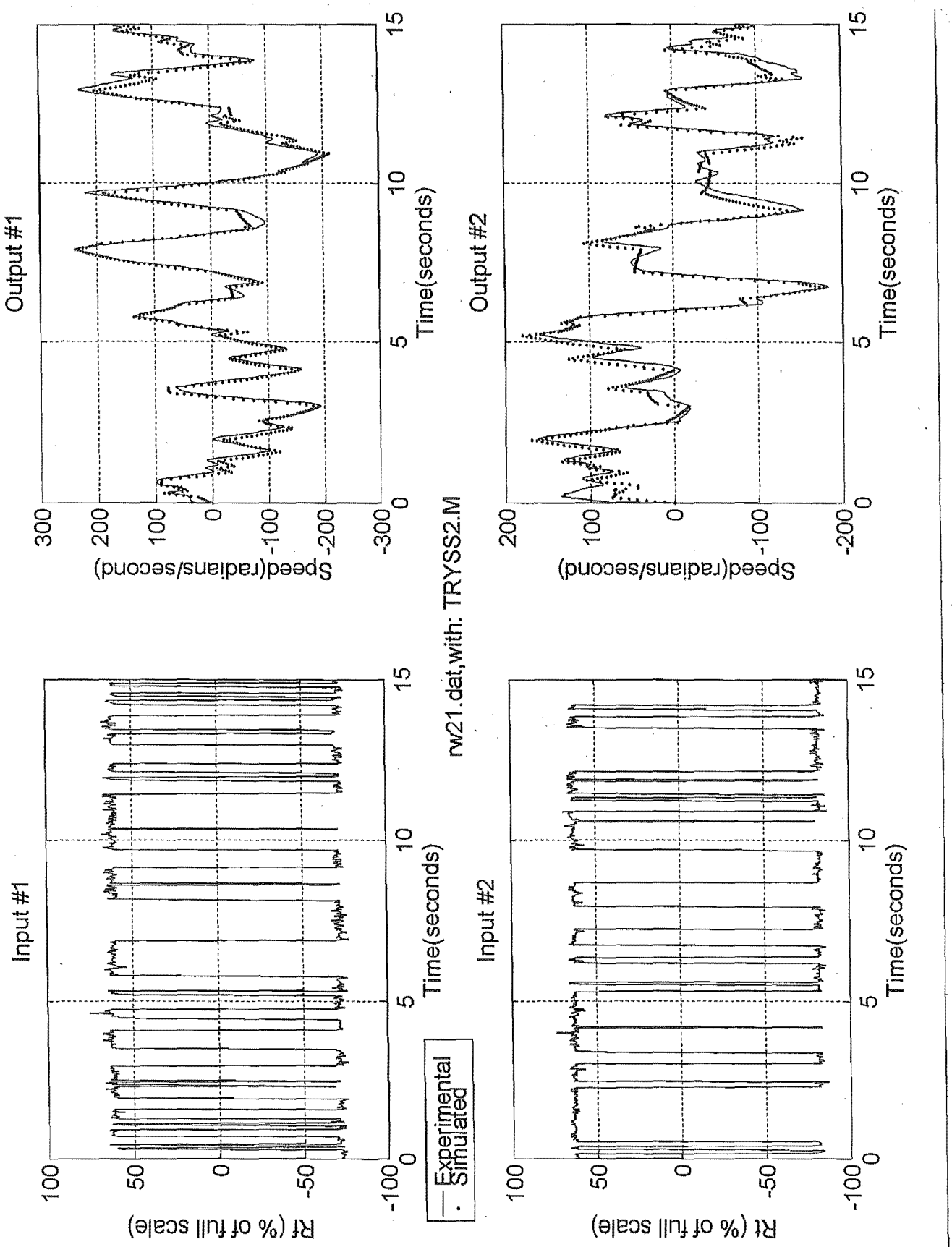
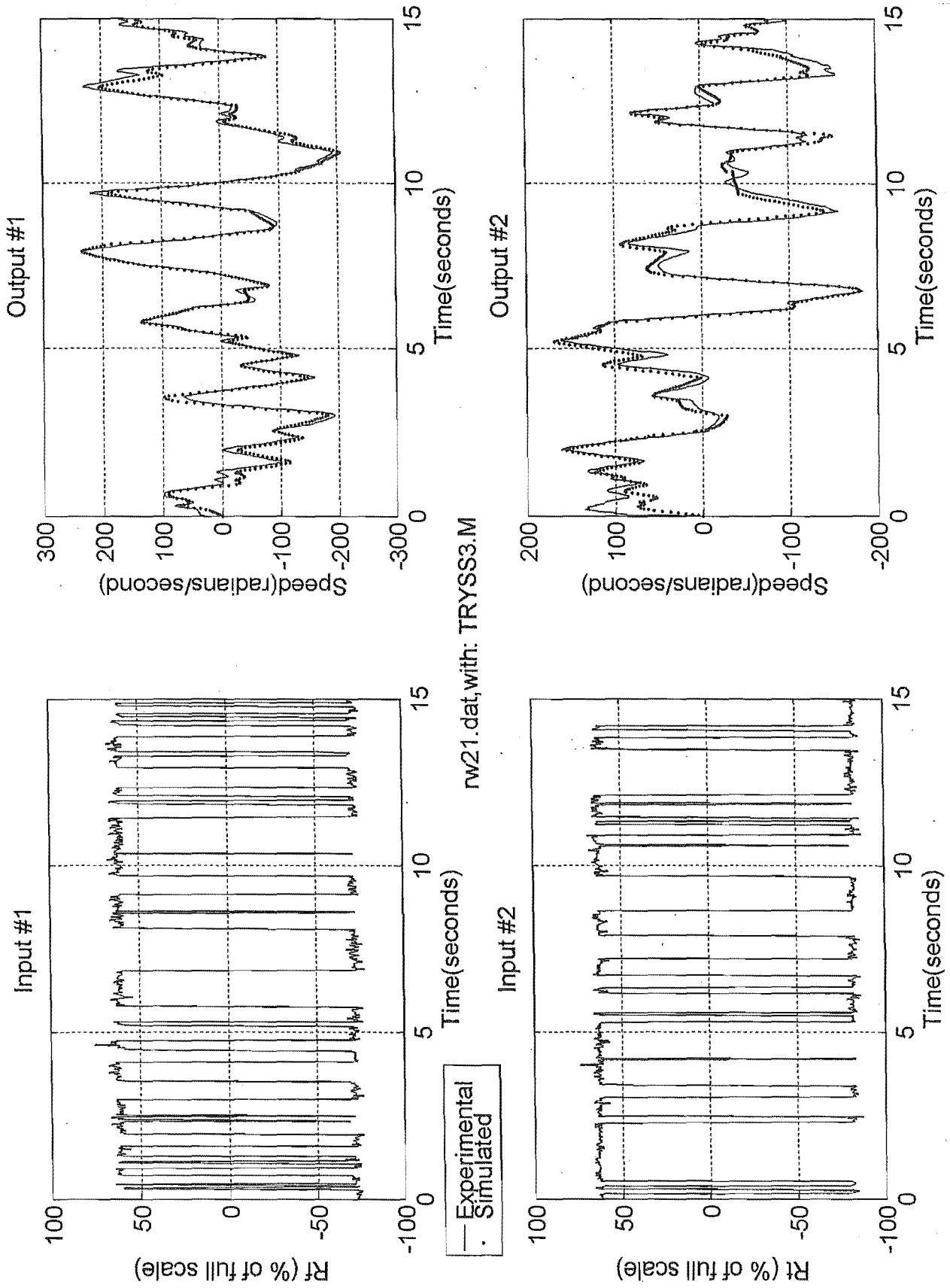


Fig. 5.6 rw21.dat with: TRYSS3.M, DCMC model 2-II



6. RESULTS AND COMPARISONS OF SIMULATIONS AND LABORATORY TESTS

6.1 Initial Conditions For AGV Laboratory Testing

The AGV has a limited sensor range (< 150 mm), and the laboratory space available to perform the tests was limited. Hence, at the start of each experimental run, the AGV was manoeuvred parallel to the guide wire with the sensor pair offset approximately 120mm from the guide wire. These initial conditions were effectively a step input in displacement from the guide wire combined with a step input in forward velocity after starting the control algorithm. The offset from the guide wire was chosen such that the AGV would be within the sensor range, and would not have to exhibit any 'searching' behaviour in order to find the guide wire. Such evolutionary behaviour would be difficult if not impossible to accurately model in the simulation.

6.2 Results Of Simulations

Generally, the more complex of the Matlab simulations show good correlation with the experimental results. Unfortunately, the greater the complexity of the model, the greater the time taken for the calculations.

In order to make a meaningful comparison between the simulation and the experimental results, initial conditions for the simulations were used that were similar to those used in the laboratory experiments.

6.2.1 Simple Linear Second Order Model

This model has the AGV on a flat plane free to roam in 2D. With the transformation to 2D, the minimal AGV becomes considerably more difficult to control. The AGV is directly provided with feedback of the known sensor location.

The performance of these simulations is not representative of the existing AGV system, but it does provide an indication of the performance of an AGV system with complete knowledge of the sensor location.

Figs. 6.1 and 6.2 show the performance of the AGV model with increasing proportional gain. As the gain is increased the AGV follow the guide wire (or desired path as is the case in this model) with a reduced tracking error. The model is damped and when the DC motors saturate, the model performance moves asymptotically to a minimal error with increasing proportional gain. Brief checks with gains of 2 orders of magnitude greater that the largest gain used on the AGV experimental runs confirm this. This asymptotic trend is shown clearly in fig. 6.2.

A numerical approximation to the non-linear system of equations describing this model was calculated for the given operating points using simple derivative approximation tools. These tools perturb the inputs to the model, and use the changes in the outputs to calculate an approximation to the state space matrices. Subsequently, these state space approximations were examined for observability and controllability of the model.

From the analysis, this model of the AGV proved to have all states controllable, but only two measured states with the given sensor equipment. For full feedback control of these unobservable states, more feedback information would be necessary. This feedback could be gathered either by adding more sensors, or by adding an observer (or Kalman Filter) to the control system to estimate the unknown states.

6.2.2 Complete Non-Linear Model

Figs 6.3 and 6.4 show the same characteristics as the simple linear model. The performance of the AGV simulation becomes asymptotically closer to a minimum error as the control effort is increased. This model includes the operation of the sensors as they were implemented on the actual AGV.

The plots vary from the simple linear model accordingly. When the AGV is further from the guide wire, the sensor strength is low and the model turns slowly toward the desired path. As the AGV turns to the guide wire, the sensor models show increased signal strength and the position error (as it simply the difference between the signal levels at the sensors) increases briefly until the sensors are positioned either side of the guide wire. Where the simulated AGV is nearer the desired path, the performance of the linear model and the non-linear sensor model are more similar.

6.2.3 Complete Complex Second Order model with Backlash

Fig. 6.5 demonstrates the performance of the model of the AGV complete with backlash between the gear train and the wheels. For these simulations, the backlash of the model was set to a value similar to that of the actual AGV. What is most distinct about these paths is the introduction of limit cycles that manifest themselves as oscillations in the AGVs displacement from the desired path.

With the increasing proportional gain, two features of the traces are noteworthy:

1) The AGV model attempts to reach the same asymptotic path as simpler models
2) The amplitude of the limit cycles increases, as the AGV model is given greater energy between the backlash cycles.

6.3 COMPARISON OF LAB RESULTS AND SIMULATIONS

The laboratory results and the simulations show similarities in many features of their performance. The results of the simulations are considered here in the light of the results of the experiments, and the possible reasons for the differences are discussed.

6.3.1 Comparison of the Backlash Model with the Experimental results

The measured AGV responses are compared with the simulated responses figs 6.6,6.7 and 6.8. The experimental results obtained show excellent agreement with the simulations over a 21:1 change in the system gain. However, examining the results of the lab experiments, it is immediately apparent from Run#3 (fig. 6.8) that assumption of no-slip is invalid at high feedback gain. Although its not obvious from the graph, the microcontroller was observed to be saturating the DC motors and over driving the AGV well into backlash. The AGV was being driven so hard that the differentials in left and right wheel speed would cause an exaggerated rotation that the control system would attempt to negate by driving the AGV in the opposite direction.

The backlash in the gear train has the effect of aggravating the traction conditions at the wheels. As soon as the gear train began to engage after the DC motor has been freewheeling, the wheels are accelerated abruptly, lose traction and spin. The spinning wheels have less tractive effort on the ground. Thus the AGV increases speed to accommodate for the increased error at the sensors, and establishes a limit cycle

oscillating between full speed wheel slip in either direction as the sensor oscillates across the guide wire.

The onset of the limit cycling through the AGV backlash region is aggravated by increasing the proportional gain. At lower proportional gains, (and thus lower control effort), the backlash model simulates the AGV with better accuracy. Fig. 6.9 shows a comparison of the experimental run at low proportional gain with two simulations at similar gain values. Each of the backlash simulations tracks the experimental results well near the beginning of the run, but an error is apparent in the middle phase of the run.

6.3.2 Potential Sources of Model Inaccuracy

In the simple models where the backlash and slip are not included, the higher gain models will apply more torque to the AGV wheels for a given error. In these models the torque is assumed to be transmitted to the AGV chassis via the tractive effort of the wheels. In reality this has proven to be incorrect as the wheels break-away from the no-slip condition and introduce a new state of dynamics to the system in the form of relative velocities and interaction between the wheels, the AGV chassis, and the ground. As such, the no-slip condition is a major source of inaccuracy in the models, as will be discussed.

Another source of inaccuracy is the assumption that the majority of the friction in the system is viscous. This was a necessary assumption for the sake of classical analysis, but with the availability of powerful numerical tools such as Simulink, it may be possible to include other forms of friction in the models, although this was not attempted. Apart from the results of the Falling Weight Test to determine the properties of the DC motors, there was no immediate measurable evidence of 'stiction' in the AGV system.

6.3.3 Potential Sources of Experimental Error

The primary source of experimental error is in the operation of measuring the points laid down on the paper, and in the physical layout of the experiment itself.

The paper was taped to the floor over the guide wire, and despite all efforts to the contrary, the guide wire was known to be linear only to within ±3mm. This source of

error should be reproduced in all the experimental runs, as the AGV operated over the same stretch of wire.

Also, the paper marking device may have introduced a degree of error into the results. As was mentioned, the marker had to be loose enough to fall freely to the ground under its own weight. About 2mm (±1mm) of transverse movement was observed in the end of the pen during operation. To reduce the effect of this movement, the marks on the paper trace were measured from the first contact point of each mark, under the presumption that the pen would be dropping to approximately the same position each time.

Fig. 6.1- AGV Linear Model path at varying proportional gain

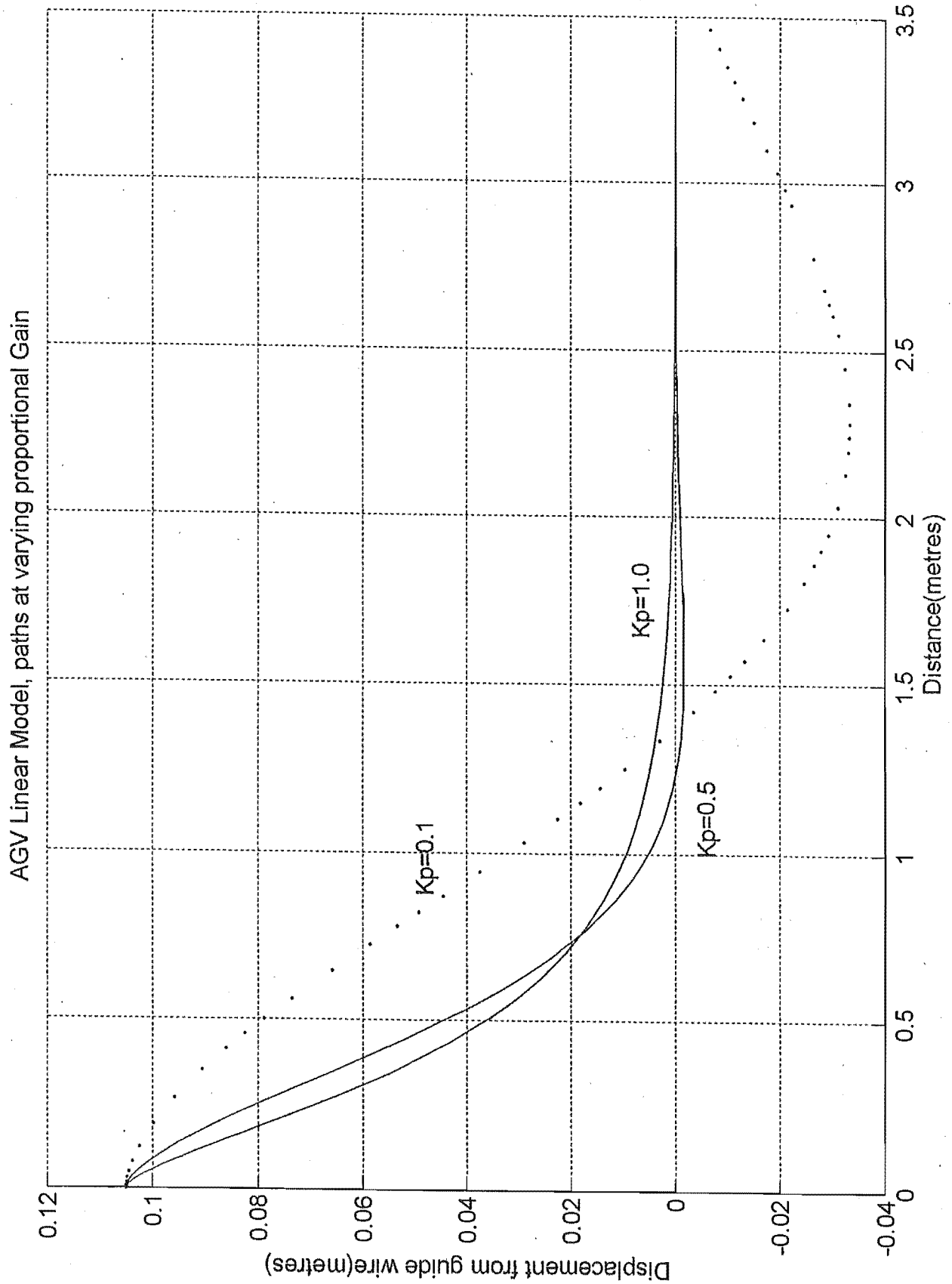


Fig. 6.2- AGV Linear Model paths with varying proportional gain

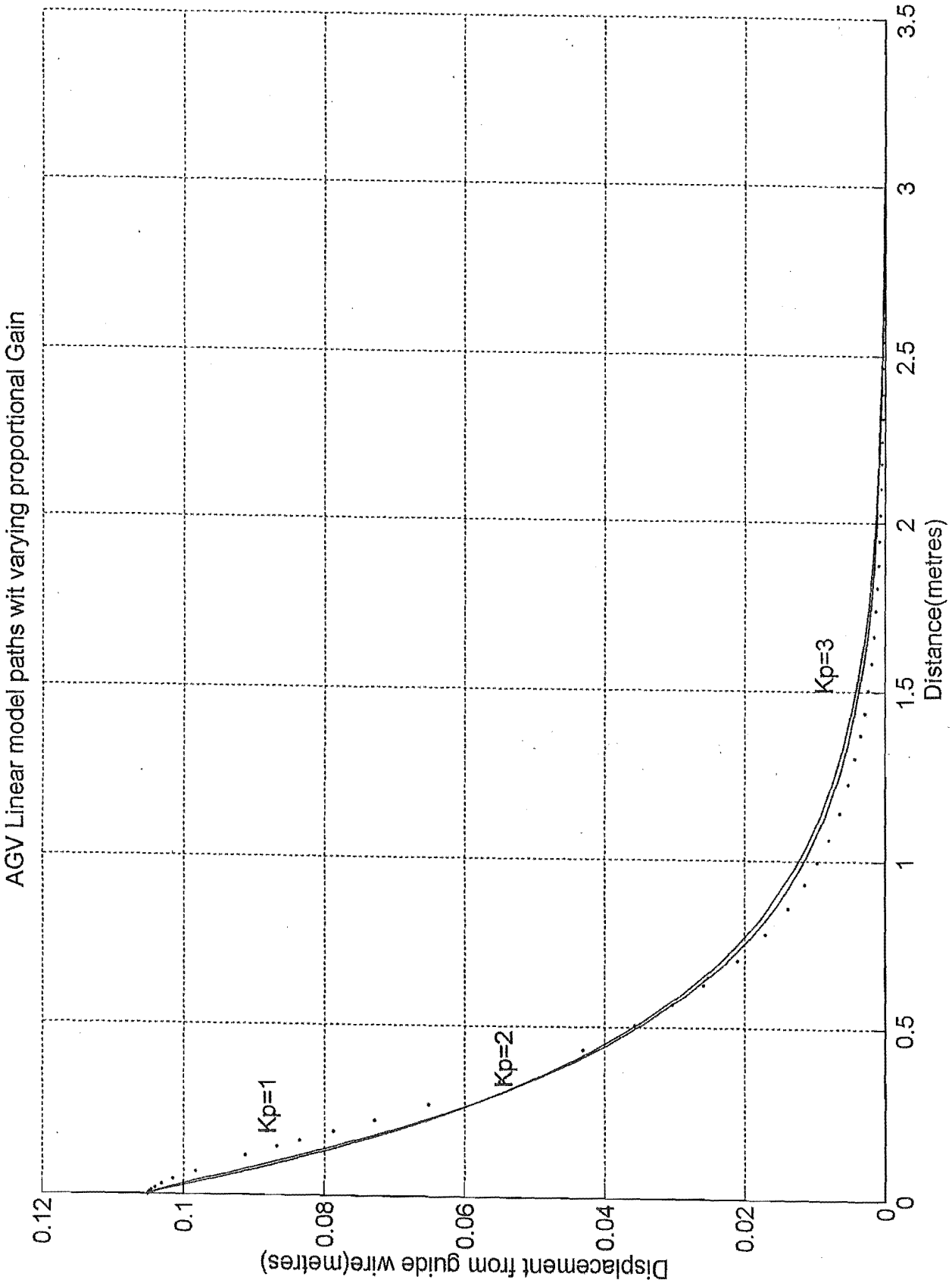


Fig. 6.3- Complete Non-linear model paths with varying proportional gain (0.1 - 1.0)

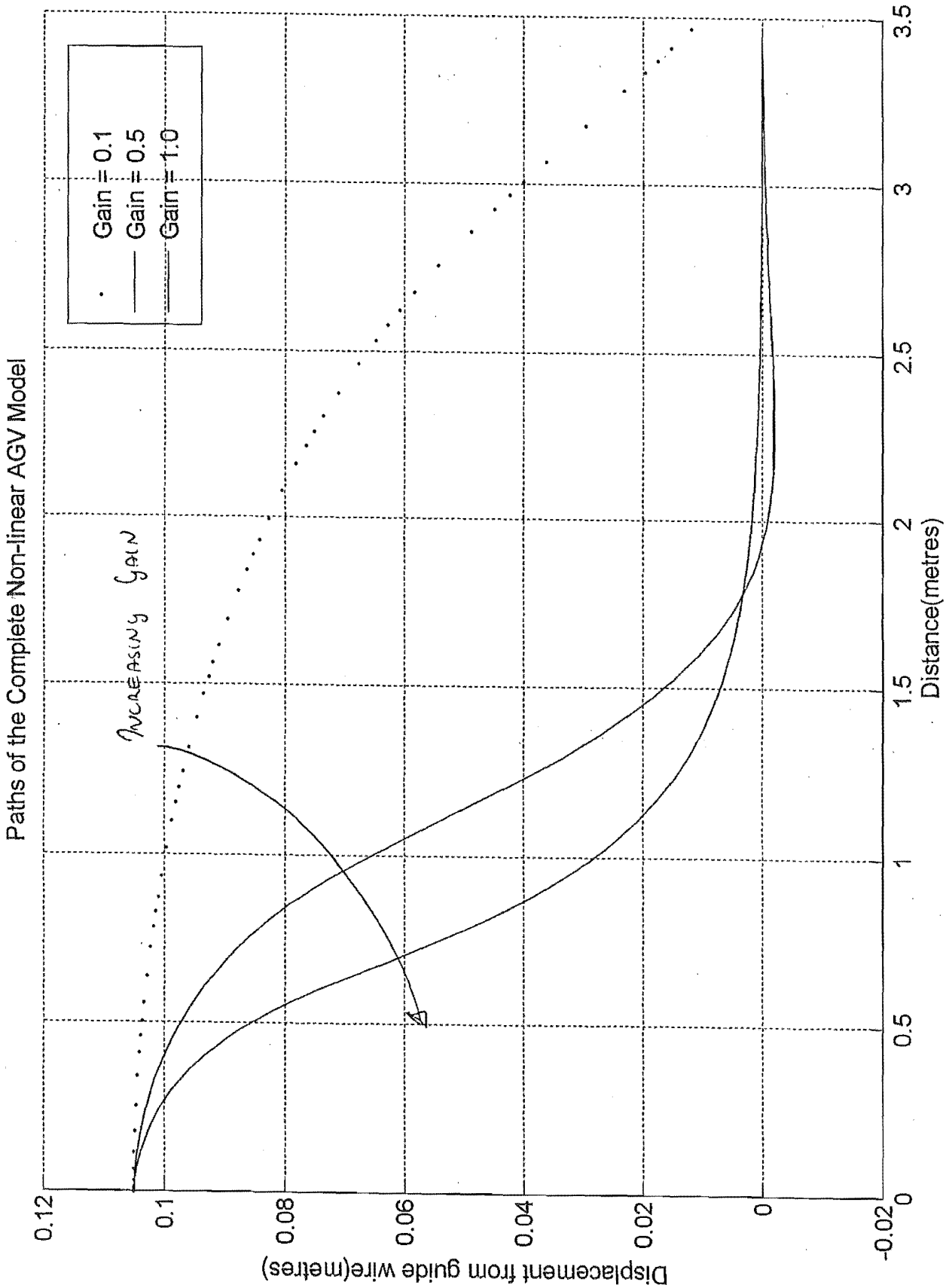


Fig. 6.4- Complete Non-linear model paths with varying proportional gain (1.5-4.0)

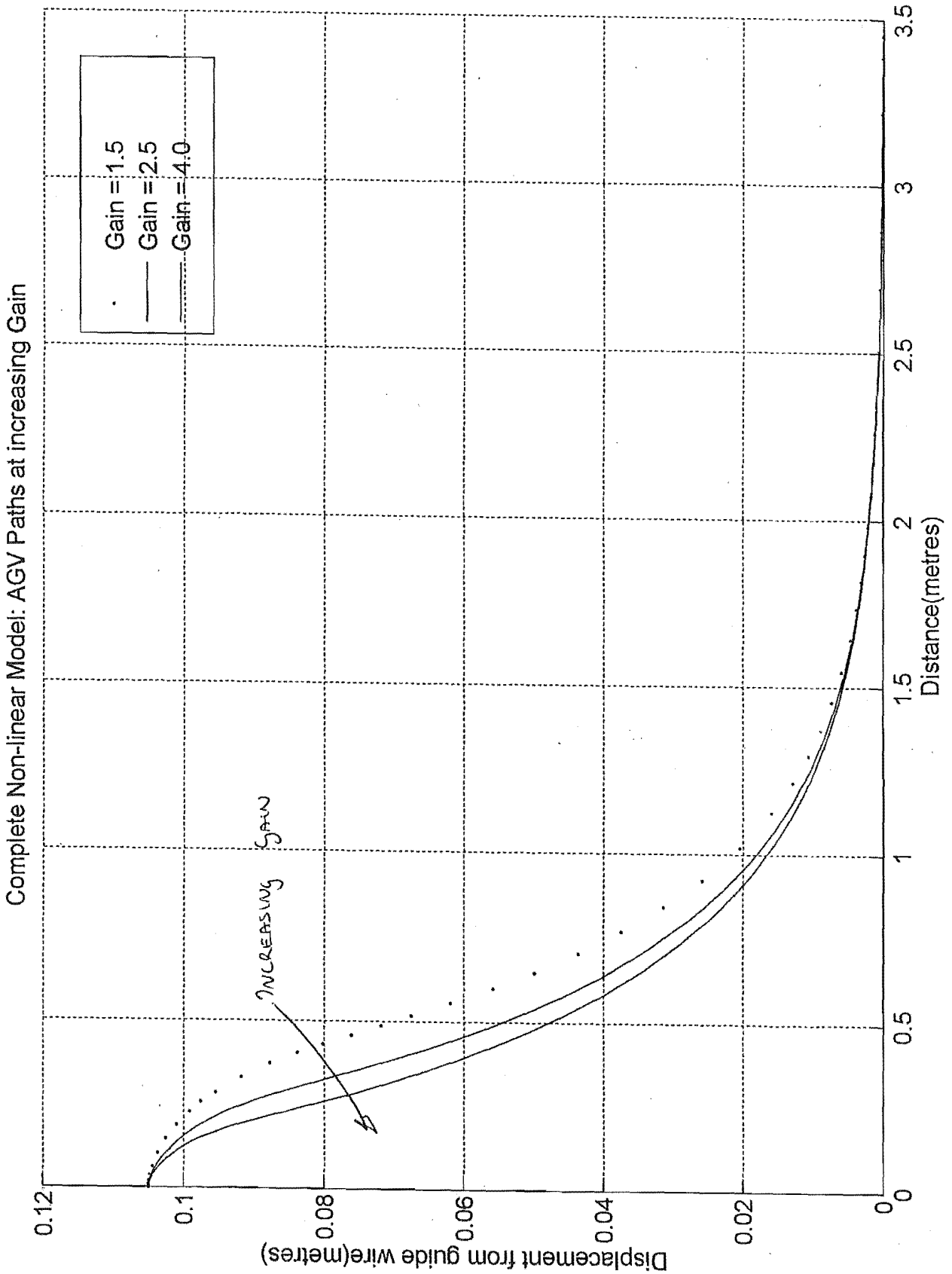
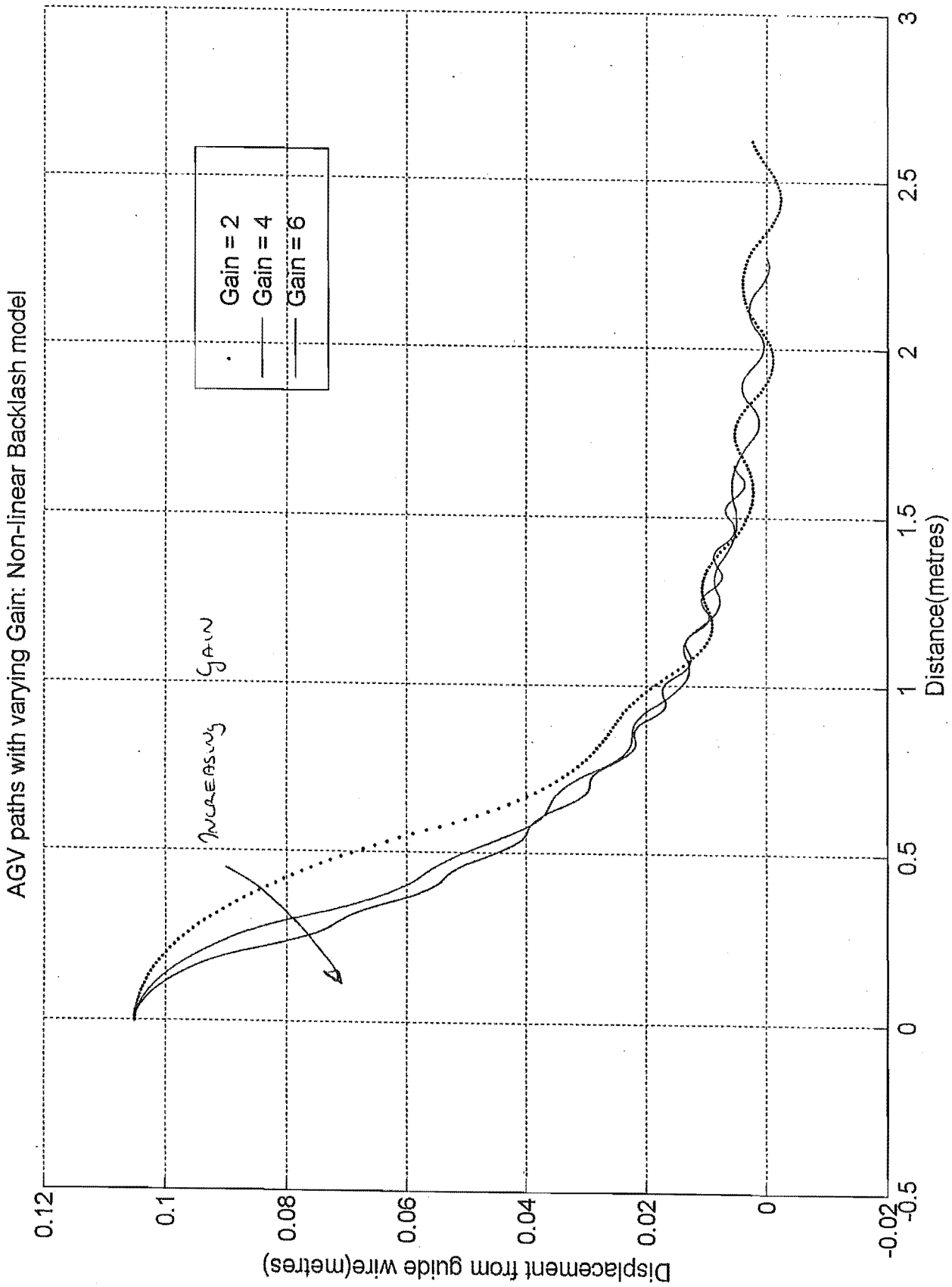


Fig. 6.5- AGV non-linear backlash model paths with varying proportional gain (2.0-6.0)



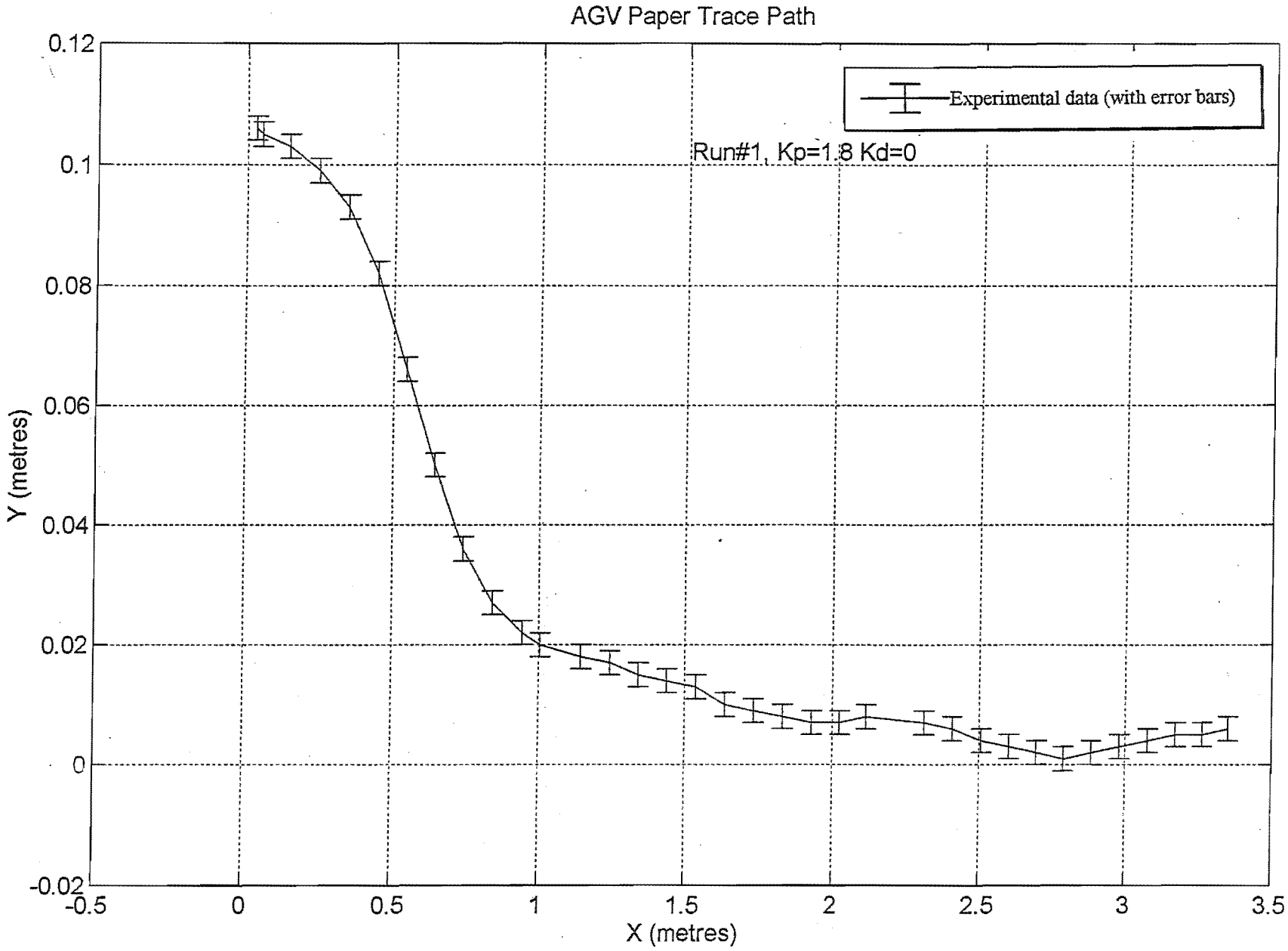


Fig. 6.6- AGV paper trace, Run #1

Fig. 6.7- AGV paper trace, Run #2

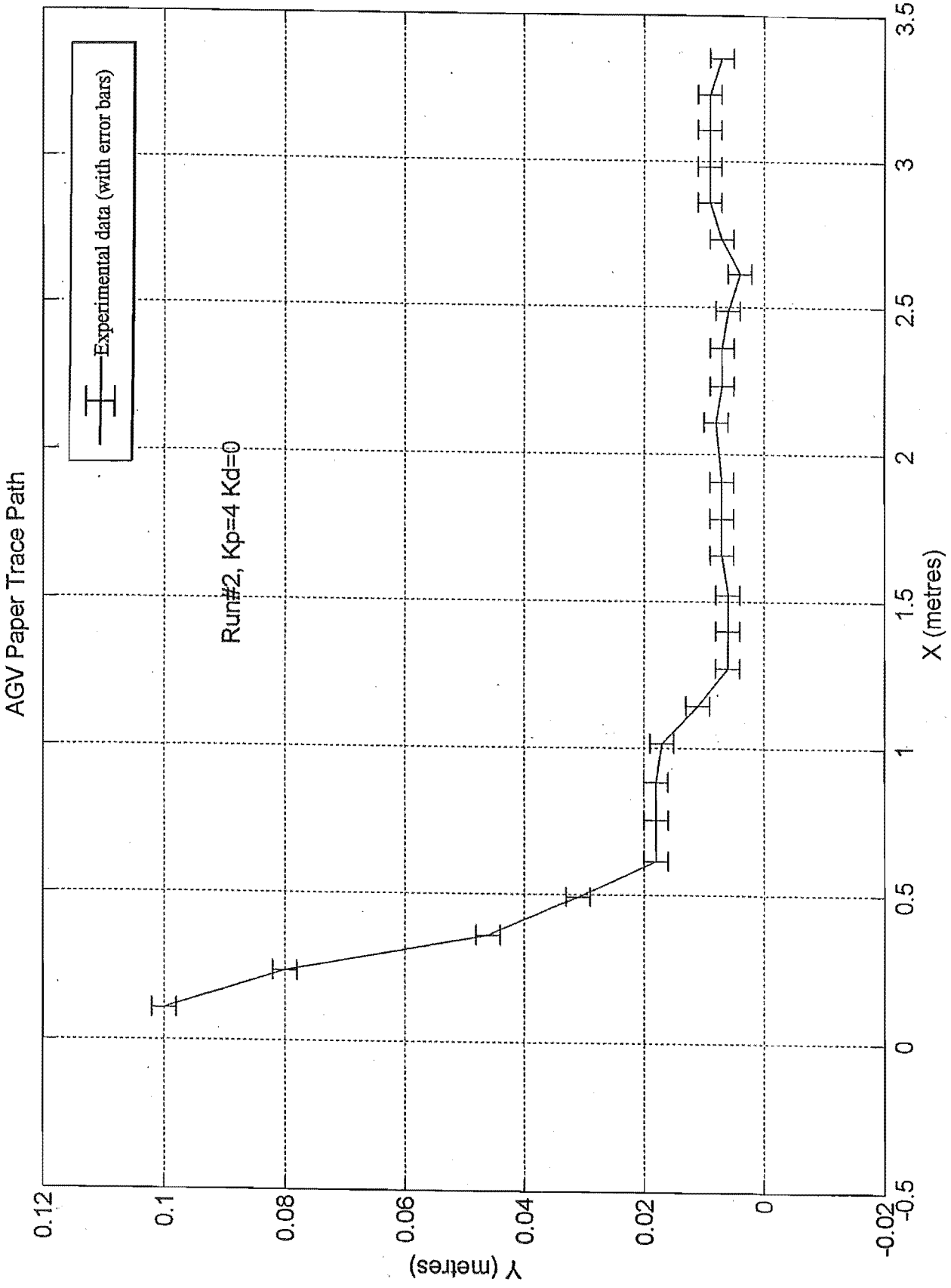


Fig. 6.8- AGV paper trace, Run #3

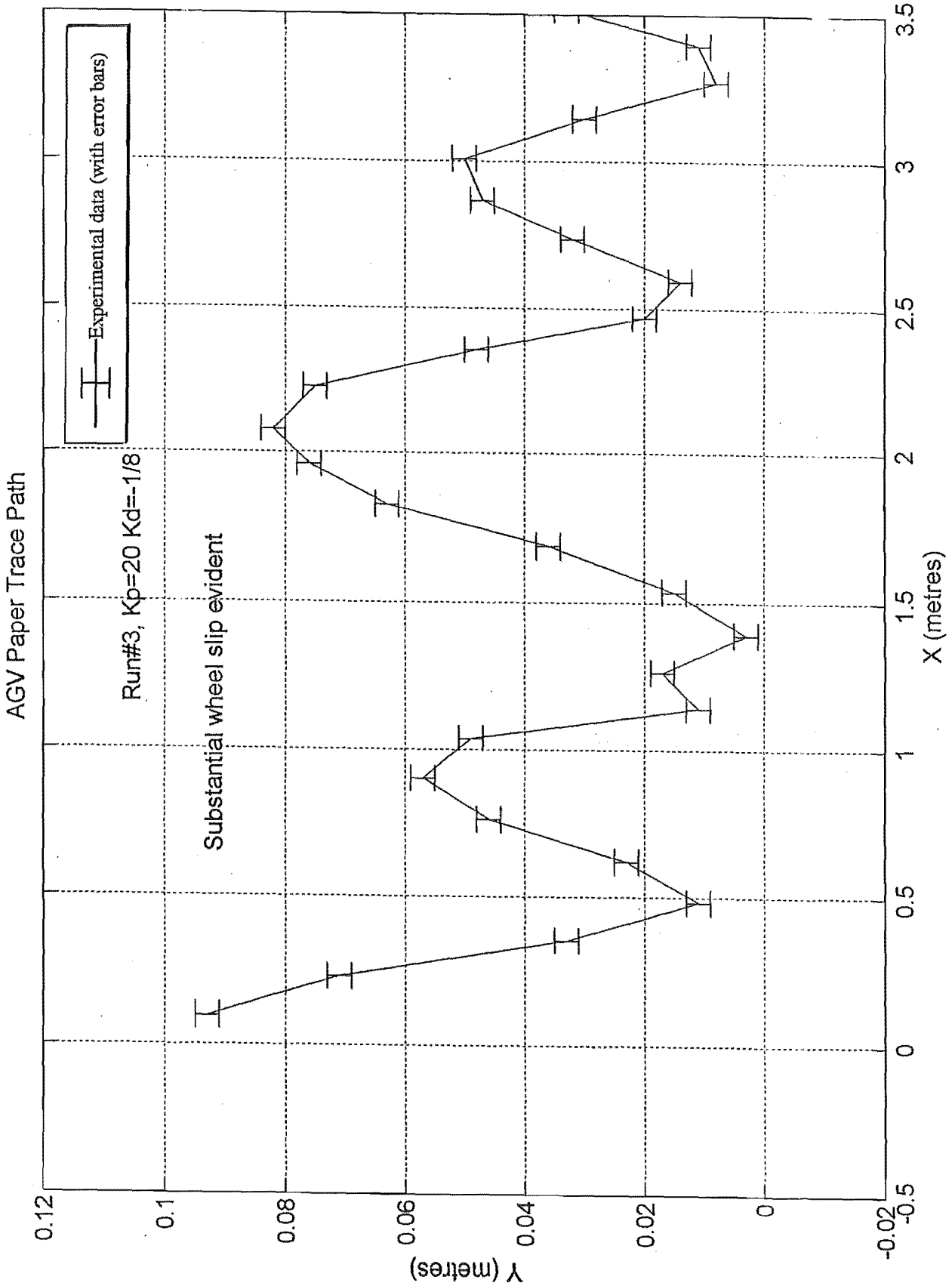


Fig. 6.9- Comparison of the experimental results to the AGV backlash model

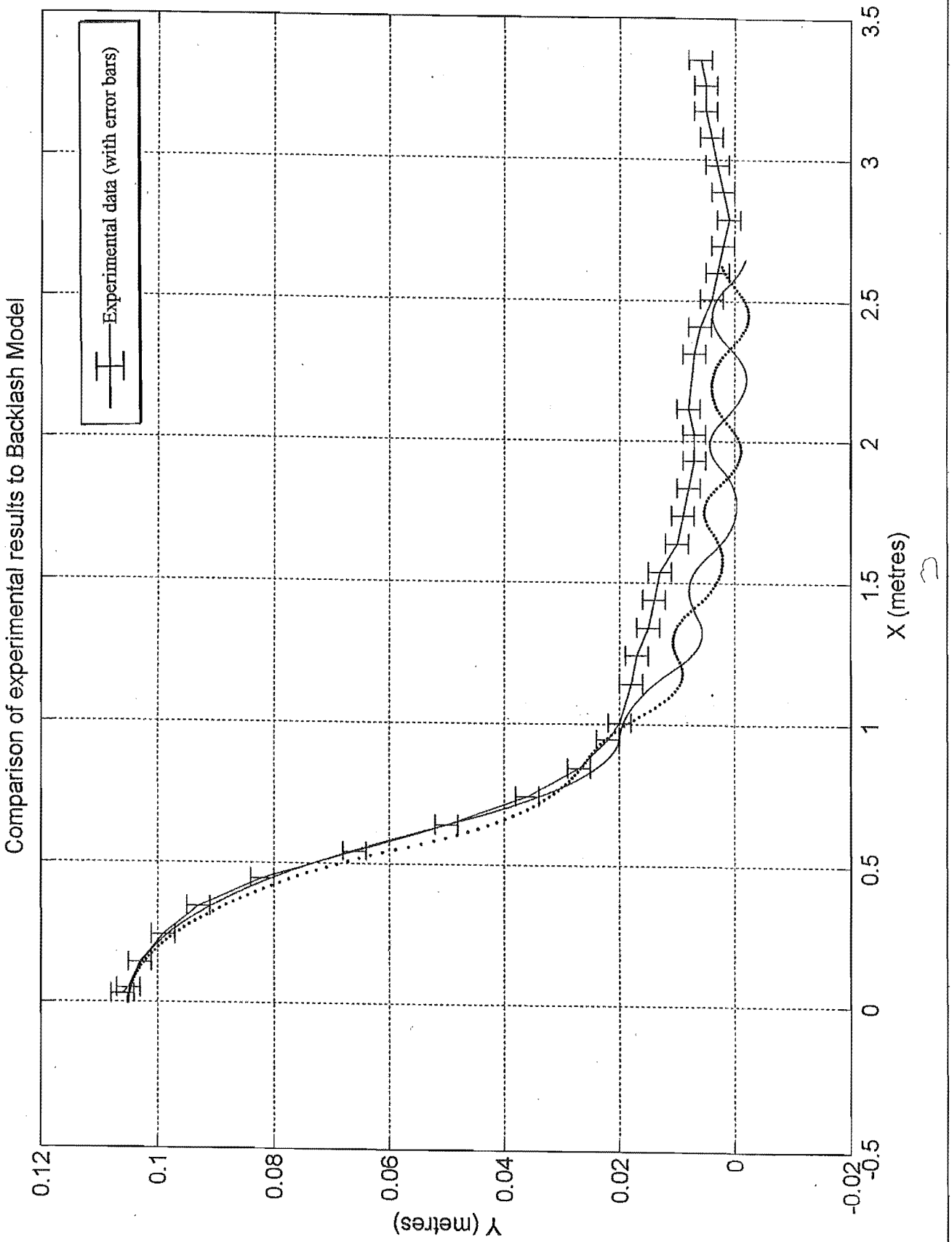
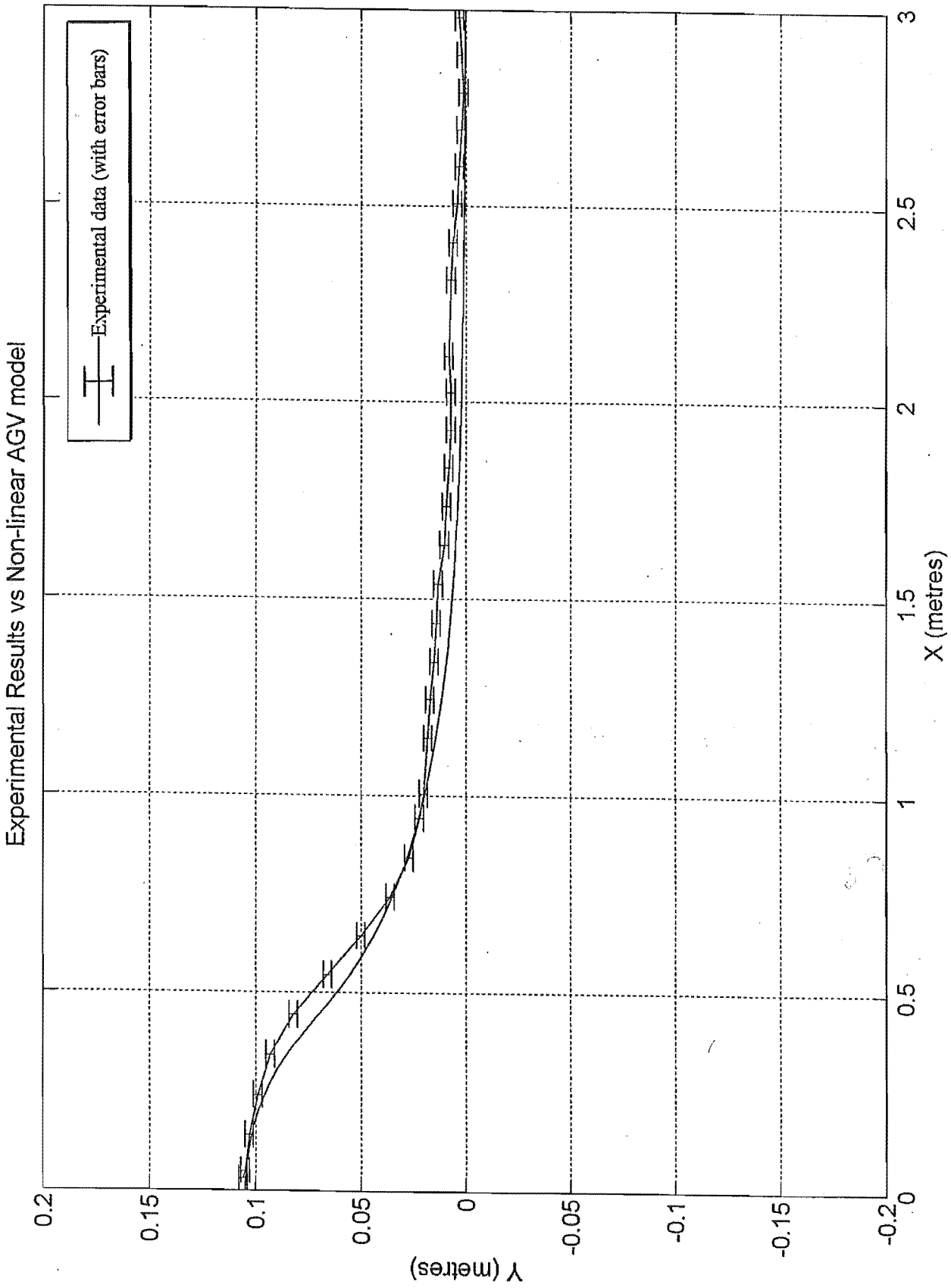


Fig. 6.10- Comparison of the experimental results to the AGV Non-Linear model



7. CONCLUSIONS

The objective of this thesis was to develop models of the AGV to examine the possibility of using the minimal AGV in situations where it has to travel in reverse. This is analogous to the "Truck Backing" problem where the driver has to manoeuvre a trailer into a space by reversing the truck. In the case of the minimal sensor AGV, the lack of observable states would make the task more like backing a trailer with no knowledge of the location of the trailer, only the position of the towbar.

The models presented in this thesis do not accurately represent the AGV at the limits of traction. However they do demonstrate a good correlation for low performance AGVs in non-rigorous conditions, or well loaded AGVs on good traction surfaces.

In terms of the minimal sensor AGVs, the UOC AGV needs development of the mechanical systems as much as the control systems. The reduction of backlash in the gear train would increase the performance envelope, as it would reduce the likelihood of traction loss under heavy control effort. From the analysis of the simple linear sensor case, the AGV is controllable in all states, but unknown in all but one, the guide wire sensor location.

The number of additional sensors to add to the AGV would be determined by the performance requirements of the target AGV system:

- The addition of a guide wire second sensor pair at the back opposite the existing pair would make the AGV bi-directional, and eliminate the reversing problem as it could always be going forwards simply by using measurements from the 'leading' pair of sensors.
- More adventurous application of the AGV requires greater investment in sensors. To control the AGV near the limits of traction, it is necessary to include some measure of the wheel speed at the very least. This would mean the addition of two angular speed transducers on the DC motors or drive wheels. A simple solution would be the addition of a 'chopper disc' to give some velocity feedback, as they are fairly simple to interface with digital controllers. Alternatively, brushless DC motors contain this information as part of the commutation command requirements.

- Further, to completely control slip, it would be necessary to sense the relative motion of the AGV with respect to the ground. This would be possible by using an encoder on one of the castors, to indicate the actual speed of the AGV. This combined with wheel speed sensors would give a good indication of the state of the AGV at all times, but would increase the cost of the unit markedly.

Two improvements to make to the UOC Mk-II AGV would include the addition of softer compound tyres for use on hard, painted surfaces, and the design of a gear train with lower backlash. One possibility would be the design of a direct drive Brushless DC motor for AGV and wheelchair applications. In terms of a minimal sensor AGV, this would virtually eliminate the backlash effects but it would raise the complexity of the control system significantly. This option is however under evaluation by wheel chair manufacturers and will probably be available for future evaluation.

8. REFERENCES

[1] Steve Hampson, 1995, Non-linear predictive control of a hydraulic actuator, *PhD Thesis, University of Canterbury, Christchurch, New Zealand.*

[2] David B Reister Francois G. Pin, 1994, Time optimal Trajectories for Mobile Robots with Two Independently Driven Wheels, *The International Journal of Robotics Research*, pp 38-53

[3] Nilanjan Sarkar, Xiaoping Yun, Vijay Kumar, 1994, Control of Mechanical Systems with rolling Constraints: Application to dynamic Control of Mobile Robots, *The International Journal of Robotics Research*, pp 55-69

[4] E. K. Boukas and J. P. Kenne, 1992, Autonomous Vehicle: A comparative Study of Control Algorithms, *European Joint Conference on Engineering Systems Design and Analysis*, pp 183-187

[5] A. Hamdy and E. Badreddin, 1992, Dynamic Modelling of a Wheeled Mobile Robot For Identification, Navigation and Control, *Robotics and Flexible Manufacturing Systems*

[6] W. Drodzd and H. B. Pacejka, 1991, Development and Validation of a Bond Graph Handling Model of an Automobile. *Journal of the Franklin Institute,* Vol-328 pp 941-957

[7] Z. Qu, D. M. Dawson, J. F Dorsey, 1994, A New class of Roust Control Laws for Tracking of Robots. *The International Journal of Robotics Research*, Vol 13 No 4 pp 355-363

[8] H. B. Pacejka, 1985, Modelling Complex Vehicle systems Using Bond Graphs. *The Journal of the Franklin Institute,* Vol 319 No 1 pp 67-81

[9] Dean Karnopp, 1979, Bond Graphs in control: Physical State Variables and Observers. *The Journal of the Franklin Institute* Vol 308, No 3 pp 219-234

[10] B. W. Johnson, J. H. Aylor, 1985, Dynamic Modelling of an Electric Wheelchair. *The IEEE Transactions on Industry Applications,* Vol 1A-21 pp 1284-1293

[11] Y. Y. Cha, D. G. Gweon, 1994, Real Time control using explicit dynamic solutions of a two-motion-modes mobile robot. *Proceedings of the institute of Mechanical Engineers,* Vol 208 pp 157-167

[12] L. Borenstein, Y. Koren, 1987, Motion Control of a Mobile Robot Platform. *The Journal of Dynamic Systems, Measurement, and Control,* Vol 109 pp 73-79

[13]H Imai,K Fujiwara, Y. Kawashima, 1987, Advanced Automated Transport System. *Automated Vehicle Systems-International Trends in Manufacturing Technology*

[14]Kai Lui, F. L. Lewis, 1988, Control of a Mobile Robot with On-board Manipulator. *Robotics and Manufacturing - Recent Trends in Research and Applications, ASME Press,* Vol 4 pp 539-547

[15]Lennart Ljung, 1986-1991, Matlab System Identification Toolbox. *The MathWorks Inc.*

[16]Lennart Ljung, 1987, System Identification-Theory For the User. *Prentice Hall*

[17]Derek Ward, 1996, Design of a robotic finger, *ME thesis, University of Canterbury, Christchurch, New Zealand.*

[18]H. Overbye, 1989, Automated Guided Vehicle Mechanical Design. *Bachelor of Engineering Project Report #38*

[19]R.J Kerr, 1989, Automated Guided Vehicle Electronic Hardware Design. *Bachelor of Engineering Project Report #22*

[20]Julian Snelder, 1989, Automated Guided Vehicle System Software. *Bachelor of Engineering Project Report #46*

[21]G.R. Dunlop and J.D. Murphy, 1993, A universal pulse processing board for instrumentation and machine control, *Proc. NELCON'93, Auckland, New Zealand,* Vol 1 pp171-178.

[22]Alan Blundell, 1982, Bond Graphs for Modelling Engineering Systems, *Ellis Horwood*

[23]The Math Works Inc, 1992, Simulink; Dynamic System Simulation Software - Users Guide, *The MathWorks Inc.*

[24]Phillips Semiconductors, 1994, 80c51 based 8-bit Microcontrollers, *Phillips Semiconductors.*

[25]Intel Corporation, 1981, MCS-51 Family of Single Chip Microcomputers- Users Manual, *Intel Corporation.*

APPENDIX A: AGV MICROCONTROLLER 'C' CODE

AGVAD.C

```
/* IMPLEMENTATION MODULE AGVAD;
      Retrieves the values of the ADC from the Inputs in round robin
fashion
      and store the resultant values in the appropriate locations.
      It also enables the ADC to be used for other purposes by checking
      see if the AD conversion was initiated by these routines  */

#include "AGVAD.H"
#include <80c552.h>

/* VAR */
/* public vars */
unsigned char SensorValue[2];

/* Functions */

void InitADC(void)
{
      /* Clear the AD control register */
      ADCON =0x00;
};

void GetSensorValues(void)
{
      InitADC();

      /* Set ADC to get Channel S1 */
      ADCON = ChanS1;

      /* Start Conversion */
      ADCON |= 0x08;

      while (!(ADCON & 0x10)); /* Wait for an answer */

      /* Store value in S1*/
      SensorValue[0] = ADCH;

      /* Clear ADCI Flag */
      ADCON ^= 0x10;

      /* Set ADC to get Channel S2 */
      ADCON = ChanS2;

      /* Start Conversion */
      ADCON |= 0x08;
      while (!(ADCON & 0x10)); /* WAIT FOR ANSWER */

      /* Store value in S2*/
      SensorValue[1] = ADCH;

      /* Clear ADCI Flag */
      ADCON ^= 0x10;

};

/* END AGVAD.*/
```

AGVCONT.C

```
/* IMPLEMENTATION MODULE AGVCONT; */
/* This Module calculates the control signal to pass to AGVDrive
      from the position of the AGV [module AGVPos] */
#include "AGVCONT.H"
```

```
#include <80c552.h>


/* VAR */
/* IMPORT Xpos from AGVPOS */
      extern int Xpos;
/* X = an array of previous X positions */
      far int X[4];
/* U = an array of previous Control Values */
      far    ControlValue U[4];

      far    Gain Kpt,Kit,Kdt;
/*static*/   ControlValue R;
/*static*/   ControlValue U0;


/*CONST*/
#define      ForwardVelocity  100

void InitControl(void)
{
      Kpt.num = -1;
      Kpt.den = 20;

      Kdt.num = 1;
      Kdt.den = 8;

      Kit.num = 0;
      Kit.den = 1;

};


void GetU(void)
{
      /* Rotate the X values along one in the X array */
      X[3] = X[2];
      X[2] = X[1];
      X[1] = X[0];
      X[0] = (Xpos);

      /* Rotate the U values along one in the U array */
      U[3] = U[2];
      U[2] = U[1];
      U[1] = U[0];
      U[0] = U0;

      U0.t = Kpt.num*(R.t - X[1]) / Kpt.den + Kdt.num*(X[0]-X[1])/Kdt.den;
      U0.f = R.f;

};
```

AGVDRIVE.C

```
/* IMPLEMENTATION MODULE AGVDRIVE;*/
#include "AGVDRIVE.H"

#include <80c552.h>

/* VAR */
#define PWMForward  PWM1
#define PWMTurn     PWM0

bit DriveDisabled @ P4_BADDR;

/* CONST */
#define  HalfDuty   0xff/2
#define      ShortestPeriod   0x01
#define        LowPWM0    20
#define        HighPWM0   236
#define        LowPWM1    20
#define        HighPWM1   236

void AllStop()
      /* Set the Motors to stop */
{
      /* Set the PWM to half duty cycle */
      PWMForward = HalfDuty;
      PWMTurn = HalfDuty;
};


void InitDrive()
      /* Initialise the AGV Drive system, via the PWM pins and Port 4*/
{
      /* Disable Drive */
      DriveDisabled = TRUE;
      /* Set the PWM Period Prescaler*/
      PWMP = ShortestPeriod;
      /* Set the PWM to half duty cycles */
      AllStop();
};

void Drive(int Forward,int Turn)
{
      static far unsigned char pt0;
      static far unsigned char pt1;

      if (!DriveDisabled)
      {
            /* Calculate the temporary PWM values pt0,1*/
            /* It is inadvisable to perform calculations on the PWM
                  Ports directly, as the DCMC will switch off power
                  to the motors if the PWM goes outside the recognised
                  boundaries.  Therefore it is necessary to use temporary
                  variables */
            pt0 = (unsigned char)(Forward+ HalfDuty);
            pt1 = (unsigned char)(Turn - HalfDuty);

              /* Check the Ranges of new PWM values */
             if (pt0 > HighPWM0)  pt0 = HighPWM0;
             if (pt1 > HighPWM1)  pt1 = HighPWM1;
             if (pt0 <  LowPWM0)  pt0 =  LowPWM0;
             if (pt1 <  LowPWM1)  pt1 =  LowPWM1;

             /* Assign the Drive values to the PWM ports */
             PWMForward = pt0;
             PWMTurn = pt1;
      }
      else {
            AllStop();
      };
```

```
};

/* END AGVDRIVE.*/
```

AGVINT.C

```
/* IMPLEMENTATION MODULE AGVINT;*/
#include "AGVINT.H"
#include <80c552.h>
#include <intrpt.h>

/* ???FROM AGVDRIVE IMPORT DriveDisabled>????; */
extern bit DriveDisabled;

/* set the INTERRUPT VECTOR */

ROM_VECTOR(EXTI0,BumperInterrupt);

/* function implementation */
void InitAGVInterrupts(void)
{
/* Initialise AGV interrupts EX0 */
      EX0 = TRUE;
/* enable global interrupts */
      EA = TRUE;
};

interrupt void BumperInterrupt(void)
{
/* Emergency stop routine - set PWM Channels to 50% and
                          - CLEAR Enable pin to DCMC */
PWM0 = 0xff/2;
PWM1 = 0xff/2;
DriveDisabled = TRUE;
EX0 = FALSE;
}

/* END AGVINT.*/
```

AGVLOOP.C

```
/* These functions use the Timed Control Loop using Timer2*/
/* T2 is set as a FRC and an interrupt is generated off a
      T2 compare.  At each interrupt it is incremented by a
      value (ControlLoopPeriod) and re-loaded. */

/*    NOTE: for this part of the program to operate properly,
      the vector for T2's compare must be set to point to the
      StubControlLoop */

/* includes */
#include "agvloop.h"
#include <80c552.h>
#include <intrpt.h>

/* constants */
ROM_VECTOR(CMP2INT,StubControlLoop);
/* set CMI2 to the control Loop */

/* variables */
unsigned char ControlLoopPeriod;
void far (*ControlLoopFunc)(void);

/* function prototypes */
void InitControlLoop(void);
void StartControlLoop(unsigned char,void far (*)(void));
void StopControlLoop(void);
interrupt void StubControlLoop(void);
```

```
/* function implementation */

void InitControlLoop(void) /* Set up T2 and it's interrupt */
{
      /* Set T2 as a Free Running Counter with a 16bit overflow
            period of around 500ms , fosc/12 + 1/2 prescaler*/
      TM2CON = 0x04;

      /* ensure that the interrupts are off */
      ECM2 = FALSE;
      CMI2 = FALSE; /* and that the flag is not set...*/
};


void StartControlLoop
(unsigned char newControlLoopPeriod,
                                                      void far
(*newControlLoopFunc)(void))
/* Start T2's compare interrupt -
      and load the variable which will point to the Control Loop Itself*/
{
      /** set up the Control Loop Variables */
      ControlLoopPeriod = newControlLoopPeriod;
      ControlLoopFunc   = newControlLoopFunc;

      /* Set up the compare registers */
      CML2 = 0xFF;
      CMH2 = TMH2 + ControlLoopPeriod;

      /* enable compare interrupt */
      ECM2 = TRUE;

      /* start timer */
      TM2CON |= 0x01;

      /* enable interrupts */
      ei();
};


void StopControlLoop(void)  /* Stop T2's interrupt */
{
      /* disable compare interrupt */
      ECM2 = FALSE;
      /* turn off timer */
      TM2CON &= 0xFC;
      /* clear flag */
      CMI2 = FALSE;
};


interrupt void StubControlLoop(void)
{
      /* clear the T2 compare flag and add the period
       to the compare register*/
      CMI2 = FALSE;
      CMH2 += ControlLoopPeriod;

      /* call the Control Loop Function */
      (ControlLoopFunc)();
};
```

AGVPOS.C

```
/* V1.4 - V1.3 converted from MODULA2 to C */
/* Calculates the position of the AGV's Sensor from the sensor inputs
      incident at the AD channel */
```

```
#include "AGVPOS.H"
#include <80c552.H>
#include <stdlib.h>

/* ???FROM AGVAD  IMPORT SensorValue; ???*/
extern unsigned char SensorValue[2];

/* defines */
#define S SensorValue
#define      S1max 80
#define      SensorMinimum  4

/* public vars */
int Xpos;
bit LostWire;
bit signX;


/*private vars */
      unsigned char gS; /* Greater Sensor value */
      unsigned char lS; /* Lesser Sensor Value */

/* TYPE */
typedef unsigned char LUColumn [21];

/* CONST */
 //    LUColumn LuX= {0,6,13,19,25,32,38,45,51,58,64,70,
 //                  77,83,90,96,102,109,115,121,127};
 //
 //    LUColumn LuS2= {80,121,151,186,214,237,240,226,210,164,
 //                  132,109,85,67,51,41,32,25,19,14,10};
 //
 //    LUColumn LuS1= {80,70,55,40,30,21,14,11,
 //                  7,5,3,2,1,1,1,0,0,0,0,0,0};

/* function implementation */

void GetXpos(void)
{
/*VAR*/
//     char i;

 //    Commented out all the 'lookup table' stuff,
 //    let's have a go a simple proportional control
 //    Using the sensor level differential...

       gS = max(S[0], S[1]);
       lS = min(S[0], S[1]);

       /* Decide if the wire has been Lost */
       if ((gS < SensorMinimum) && (lS < SensorMinimum))
       {
             LostWire = TRUE;
       }
       else
       {
             LostWire = FALSE;
       };

       /* Deduce the sign of X */
       if (S[0] < S[1])
       {
             signX = TRUE;
       }
       else
       {
             signX = FALSE;
       };

 //
 //    /* Decide which variable, lS or gS, will
 //          be used on the lookup table */
```

```
//    if (gS < S1max)
//    {
//          for(i = 0; i <= 20 ; i++)
//          {
//                if (LuS2[i] > gS)
//                {
//                      Xpos = LuX[i];
//                };
//          };
//    }
//    else
//    {
//          for (i = 0; i <= 20; i++)
//          {
//                if (LuS1[i] > lS)
//                {
//                      Xpos = LuX[i];
//                };
//          };
//    };
//
//    if (signX == TRUE)
//    {
//       Xpos = -Xpos;
//    }

 Xpos = S[0]-S[1];
}

#undef S
/* END AGVPOS.*/
```

AGVSER.C

```
/* IMPLEMENTATION MODULE AGVSER; */
/* simple module that reads 8 bits A/D on the 80C552
        and dumps out serial port */

#include "AGVSER.H"
#include <80c552.h>

/* CONST */
char HexTbl[] = {"0123456789ABCDEF"};

void SendP(register char Ch)
{
 /* Wait until the Serial transmit buffer is empty*/
 while (!(TI));  /* Stop bit transmitted??*/
 TI = FALSE;
 SBUF = Ch;
};

void SendHexH(register unsigned char HexB )
{
   SendP(HexTbl[HexB % 16]); /* Send hH Nibble */
};

void SendHexHH(register unsigned char HexB )
{
 SendP(HexTbl[HexB / 16]); /* Send Hh Nibble */
 SendP(HexTbl[HexB % 16]); /* Send hH Nibble */
};

void Send0xDDD(char num)
/* Send a byte to the serial port in signed Decimal Form */
{
   if ( num < 0)
   {
      SendP('-');
      num = -num;
```

```
    }
    else SendP('+');

    SendHexH(num /100);
    num = num % 100;
    SendHexH(num /10);
    num = num % 10;
    SendHexH(num);
}

void SendINT(int i)
{
        if (i < 0)
       {
                SendP('-');
                i = - i;
       }
    else SendP('+');
    SendHexH((unsigned char)(i / 10000));
    i = i % 10000;
    SendHexH((unsigned char)(i /  1000));
    i = i %  1000;
    SendHexH((unsigned char)(i /   100));
    i = i %   100;
    SendHexH((unsigned char)(i /    10));
    i = i %    10;
    SendHexH((unsigned char)(i));
};

void InitSerial(void)
{
 TMOD = 0x21;   /* TIMER 1  8 bit Autoreload Timer 0  16 bit */

 TH1 = 0x0FD; /* for 9600 baud 11Mhz */

/* SOCON = 50H;  */
 /* 8 bit UART, No Parity Timer1 baud control, Rx ON*/
 SCON = 0x50;
 TR1 = TRUE;  /* start T1 */
 TI  = TRUE;  /* SET TI-transmit interrupt flag, fool SendP */
};

void Wait10mS(void)
{
 register int DP = 1024;

 while (DP) DP--;
};

void BOL()
       /* Send Carriage return */
{
       SendP(0x0D);
       Wait10mS();
};

void NEWLINE()
       /* Send Carriage Return - Line Feed */
{
       BOL();
       SendP(0xA0);
       Wait10mS();
};

/* END AGVSER.*/
```

PLAYTIME.C

/* playtime.c */

```
/* small program for the AGV to run on the OPEN day */

/* The AGV will exhibit a simple behaviour, travelling to the
      end of the guide wire and spinning around until it re-discovers
      the wire and continues off along the wire */

#include "AGVDRIVE.h"
#include "AGVCONT.h"
#include "AGVPOS.h"
#include "AGVAD.h"
#include "AGVINT.H"
#include "AGVLOOP.H"
#include "AGVVARS.H"

#include "..\lcd\LCD.h"
#include "..\lcd\events.h"

#include <stdio.h>
#include <80c552.h>
#include <intrpt.h>

/* defines */
#define  JIFFYTIME 3000
#define      COOEE     0x60
#define  CNTRLPERIOD 0x10
#define  TURNSPEED  0x20
#define  CRAWLSPEED 0x20
#define  WALKSPEED  0x40
#define  RUNSPEED          0x65
#define  CLOSETOHOME 0x30

/* variables */
varT varKdt_num ={"Kdt.num",&Kdt.num,INT};
varT varKdt_den ={"Kdt.den",&Kdt.den,INT};
int turn;

/* function prototypes */
void InitEveryThing(void);
void ControlClosedLoop(void);
void UpdateInputs(void);
void UpdateScreen(void);
void Delay(unsigned char);

/* function implementation */

void main (void)
{
      InitEveryThing();

   /* Wait for the button press to start */
   LcdPrintAt(1,5,"-=<AGV MK2>=-");
      LcdPrintAt(2,2,"[ANY KEY TO BEGIN]");
      while (!IsEvent());
      LcdCls();

      AGVChangeVar(&varKdt_num);
      AGVChangeVar(&varKdt_den);

      DriveDisabled= FALSE;
      Delay(60);
      StartControlLoop(CNTRLPERIOD,ControlClosedLoop);

      while (TRUE)
      {
            /*-- Go to the end of the guide wire -- */
```

```
            /* Ahead slow, the AGV over the center of the wire */
            R.f = CRAWLSPEED;
            R.t = 0;

            while (!LostWire)
            {
                  UpdateScreen();
                  LcdPrintAt(1,1,"FORWARD");
            };

            /* stop 'cos we've lost the wire */
            R.f = 0;

            /*-- Spin around 'till the wire is found again --*/
            /* suspend the ControlLoop */
            StartControlLoop(CNTRLPERIOD,UpdateInputs);

            if (signX = FALSE)
            {
                  turn = -TURNSPEED;
            }
            else
            {
                  turn = TURNSPEED;
            };

            /* set turbines to speed and SPIN, no forward velocity */
            /* spin until it detects the wire */
            do {
                  LcdPrintAt(1,8,"SPINNING");
                  UpdateScreen();
                  Drive(0,turn);
            } while ((COOEE > SensorValue[0]) && (SensorValue[1] <
COOEE));

            /* and center on the wire */
            R.t = 0;
            R.f = 0;

            Kpt.num = -1;
            Kpt.den =  4;

            StartControlLoop(CNTRLPERIOD,ControlClosedLoop);

            /* until we're within COOEE */
            while ((CLOSETOHOME < Xpos) && (Xpos > -CLOSETOHOME))
            {
                  LcdPrintAt(1,2,"CENTERING");
                  UpdateScreen();
            };

            Kpt.den = 10;
      };
};

void InitEveryThing(void)
{
   /* Initialise the various systems */
   LcdInit();
   InitDrive();
   InitControl();
   InitADC();
   InitAGVInterrupts();
   InitControlLoop();
   InitEvents();
};

void ControlClosedLoop(void)
{
      GetSensorValues();
      GetXpos();
      GetU();
```

```
        Drive(U0.f,U0.t);
};

void UpdateInputs(void)
{
        GetSensorValues();
        GetXpos();
};

void Delay(unsigned char jiffs)
/* Delay a number of Jiffies - 1 Jiffy = 1/60 sec */
{
   register unsigned int j;

        while (jiffs--)
        {
                 j = JIFFYTIME;
                while(j--){;};
        };
};

void UpdateScreen(void)
{
        LcdGoto(1,2);
        LcdPrintInt((void *)(&Xpos));
        LcdGoto(6,2);
        LcdPrintByte(&SensorValue[0]);
        LcdGoto(11,2);
        LcdPrintByte(&SensorValue[1]);
};
```

AGVVARS.C

```
/* AGVVARS.C */

/* This module implements the variable adjustment program via
      the keyboard and LCD screen on the AGV */

/* It enables the user to change variables within the program
      provided they are listed in the program at compile time */

/* includes */
#include <80c552.h>
#include <stdio.h>

#include "agvvars.h"
#include "..\lcd\lcd.h"
#include "..\lcd\events.h"

/* defines */

/* variables */
char varstr[6];

/* function prototypes */

void AGVChangeVar(varT *);

/* function code */

void AGVChangeVar(varT *var)
{
      register char byte1,byte2;
      char DONE = FALSE;
      register void far *baddr;

      InitEvents();

      while(!DONE)
      {

            baddr = var->addr;

            LcdPrintAt(1,1,var->name);
            LcdPrintAt(1,14,"0x");
            LcdPrintAt(2,1,"0x100[-][+]0x1[-][+]");
            LcdGoto(16,1);

            switch (var->type)
            {
                  case INT:
                  {
                        LcdPrintInt((int *)(baddr));
                        break;
                  };
                  case BYTE:
                  {
                        LcdPrintByte((char *)baddr);
                        break;
                  };
            };

            switch (IsEvent())
            {
                  case M1:
                  {
                        *(char *)baddr -= 1;
                        break;
                  };
                  case M2:
                  {
```

```
                        *(char *)baddr +=1;
                        break;
                    };
                    case M3:
                    {
                        if (var->type == INT) *(int *)baddr -= 0x001;
                        break;
                    };
                    case M4:
                    {
                        if (var->type == INT) *(int *)baddr += 0x001;
                        break;
                    };
                    case OK:
                        DONE = TRUE;
                    case CANCEL:
                        DONE = TRUE;

                };
        }; /* end while !done */
        LcdCls();
}; /* end AGVChangeVar */
```

APPENDIX B: SUPPORT PC 'C' CODE

IDSYS.C DATA LOGGING UTILITY

```
// Here's a program to log the time history of the velocity of
// two channels of the AGV conmtrol system.

// IDSYS.C, logs data from the UPP counters and AD channels and
// stores it to a file for processing in MATLAB.

/* The user will nominate a logging period and frequency.
      The program will warn if theses are not feasible values.
      The program will present a summary of the logging data and
            prompt the user to begin input.

      The User will operate the joystick control in a p-random fashion
      The Program will periodically determine the state of the system,
            recording the DCMC input and the motor speeds */

/* LOGLEG.C : Log data points from the UPP into the PC memory for download*/

#ifndef __LARGE__
#error Compile in  large memory model to link with the PWML.lib library
#endif

#include <stdio.h>
#include <alloc.h>
#include <stdlib.h>
#include "pwm.h"
#include "\upp\upp.h"
#include <conio.h>
#include <dos.h>      /* for delay() */

/* program defines and constants */

#define UPPADDR 0x0232

/* global variables */
int *PWMdata0;
int *ADdata0;
int *PWMdata1;
int *ADdata1;

int datareadyflag;   unsigned long int numsamples;
unsigned long int sample;

char DATAFORMAT[]={"%9.5f\t%8i\t%8i\t%8u\t%8u\n"};
char TOBEGIN    []={"BEGIN"};
char TOEXIT     []={"EXIT"};
char TOCONTINUE[]={"CONTINUE"};
/* read the counter and stuff the value into a variable */
void TimerInterrupt(void)
{

      if (sample < numsamples)
      {
            // Set the AD0 scans going (hopefully)
            UPPStartAD(UPPADDR,AD0|STARTAD);
            delay(1); // delay 1 millisecond - something to do with the A_D.c
file

            //get Encoder positions
            PWMdata0 [sample]=ReadSingleEncoder(0);
```

```
            //get AD signals
            while(UPPIntStatAD(UPPADDR));
            /*    ^^^^^^^^^^^^^^^^^^^^ */
            /* this flag is broken !!! doesn't return anything meaningful*/
            delay(1);
            ADdata0 [sample]=UPPReadAD(UPPADDR,AD0);

            UPPStartAD(UPPADDR,AD1|STARTAD);
            delay(1); // delay 1 millisecond - something to do with the A_D.c
file

            PWMdata1 [sample]=ReadSingleEncoder(1);

            while(UPPIntStatAD(UPPADDR));

            delay(1);
            ADdata1 [sample]=UPPReadAD(UPPADDR,AD1);

            sample++;
            return;
      }
      else
      {
            datareadyflag = 1;
            return;
      }
}


/* Detect the existance of the UPP and the version number of the software */
void DetectUPP(void)
{
      BOARD card;
      unsigned int chipsfound;

      chipsfound=UPPFindBoard(&card);

      printf("UPP chips found: %i\n",chipsfound);
      printf("UPP software version V%i",UPPVersion());

      printf("\n[ANY KEY TO CONTINUE]");
      while(!kbhit());

}


void PressAnyKey( char *msg)
{
      if (kbhit()) getch(); // clear the buffer of a characher...
      printf("\n[PRESS ANY KEY TO %s]",msg);
      getch();
}


/* begin the main section */
void main()
{
      int    i;
      unsigned int interruptsteps;
      float  numseconds,timestep;
      FILE *outputfile;
      char outputfilename[100];
      char comment[163];
      struct date d;
      struct time t;
      unsigned long int ts;
```

```
	datareadyflag = 0;
	PWMdata0=NULL;
	ADdata0 =NULL;
	PWMdata1=NULL;
	ADdata1 =NULL;

	clrscr();
	DetectUPP();

	/* get the approximate timestep from the user */
	do
	{
		clrscr();
		printf(" -=<<Encoder and Control Signal Data logger V1.0>>=-
\n");
		printf(" Enter Timestep(ms): ");
	} while (!(scanf("%f",&timestep)));

	// warning of limited resolution due to delay()s in the interrupt
routine
	if (timestep < 5)
		{
		printf ("\nTimestep Limited to 5ms, due to Interrupt
resolution");
		timestep = 5;
		PressAnyKey(TOCONTINUE);
		}

	interruptsteps = (unsigned int) (timestep/3.25*1000);

	/* get the sample time from the user */
	do
	{
		clrscr();
		printf(" Enter Sample period(s): ");
	} while (!(scanf("%f",&numseconds)));

	/* calculate the number of samples */
	numsamples = (long unsigned int)1000*(numseconds / timestep);
	if (numsamples <= 0)
	{
		printf("Invalid negative numsamples");
		return;
	}

	PWMdata0 = (int*) malloc(sizeof(int)*numsamples);
	ADdata0  = (int*) malloc(sizeof(int)*numsamples);
	PWMdata1 = (int*) malloc(sizeof(int)*numsamples);
	ADdata1  = (int*) malloc(sizeof(int)*numsamples);

	/* bug out if it's a bad pointer allocating memory*/
	if ((PWMdata0 == NULL) || (ADdata0==NULL)
                                                || (PWMdata1 == NULL) ||
(ADdata1==NULL))
	{
		printf("error malloc'ing the data arrays");
		exit(1);
	};

	/* things start getting a bit hairy here...*/
	/* install the encoder */
	InstallEncoder(2);
	SetEncoder(0,0);
	SetEncoder(1,0);
	/* install the timed interrupt vector */
	InstallTimerInt(TimerInterrupt);
```

```
	/* purtyfy the screen */
	clrscr();
	printf("SUMMARY:\n");
	printf("Desired Timestep(ms):%f\n",timestep);
	printf("Actual Timestep(ms):%f\n",interruptsteps*3.25/1000);
	printf("NumSamples:%u\n",numsamples);
	printf("NumSeconds:%f\n",numseconds);

	PressAnyKey(TOBEGIN);

	sample =0;
	datareadyflag = 0;

	StartTimerInt(interruptsteps);

	while(!datareadyflag)
	{
		gotoxy(1,8);
		printf("SAMPLE: \t%u",sample);
		printf("/\t%u\t",numsamples);
	};

	UPPStopAD(UPPADDR);

	StopTimerInt();

	RemoveTimerInt();

	/* OK so we got the data, now to put it somewhere..*/
	do
	{
		/* get a filename */
		clrscr();
		printf("Output file name please:");
		scanf("%s",outputfilename);
		/* check to see if there is a file of that name...go back if
there is */
	}while((NULL != fopen(outputfilename,"rt")));

	outputfile = fopen(outputfilename,"wt");

	printf("Comment:");
	comment[0]=160; /* set the maximum string length */
	cgets(comment);
	printf("\n");

	getdate(&d);
	gettime(&t);

	fprintf(outputfile,"DATE:%d/%d/%i",d.da_day,d.da_mon,d.da_year);
	fprintf(outputfile," @%d:%d\n",t.ti_hour,t.ti_min);

	fprintf(outputfile,"COMMENT: %s \n",&comment[2]);

	for (i = 0; i < numsamples; i++)
	{
		char WannaSee = 1;

		float time;

		if (kbhit()) WannaSee = 0;

		time = (float)i * (float)interruptsteps * 3.25e-6;
		if (EOF == fprintf(outputfile,DATAFORMAT,
				time,PWMdata0[i],PWMdata1[i],ADdata0[i],ADdata1[i]))
		{
```

```
                printf("/nError writing data to:%s",outputfilename);
                exit(1);
        }
        if (WannaSee) printf(DATAFORMAT,
                        time,PWMdata0[i],PWMdata1[i],ADdata0[i],ADdata1[i]);
        else
        {
                gotoxy(1,1);
                printf("-=wRiTiNg tO fIlE=-");
                gotoxy(1,1);
                printf("=-WrItInG To FiLe-=");
        };

};

printf("\nData written to: %s",outputfilename);

PressAnyKey(TOEXIT);

/* all done - tidy up*/
clrscr();

/*close the file*/
fclose(outputfile);

/*free the malloc'd memory*/
free(PWMdata0);
free(ADdata0);
free(PWMdata1);
free(ADdata1);
};
```

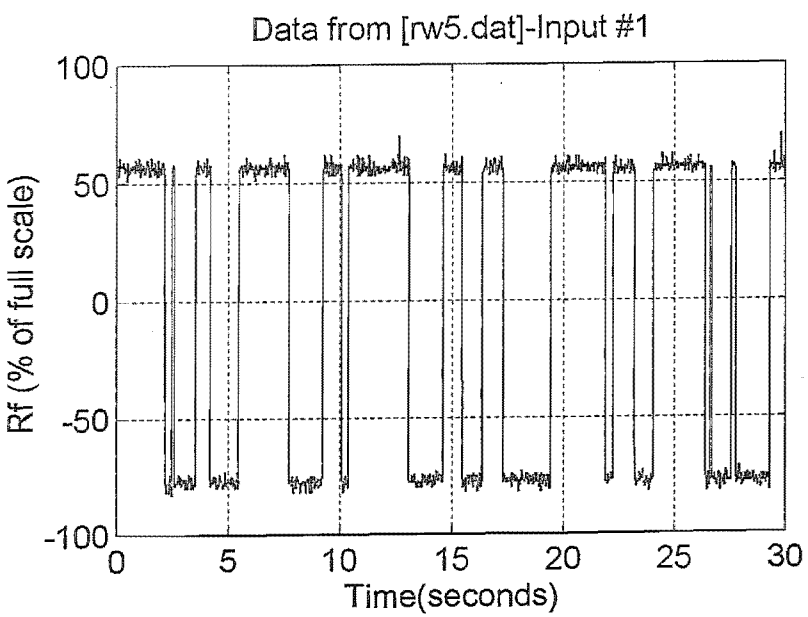
APPENDIX C: GRAPHS OF THE SYSTEM IDENTIFICATION TESTS

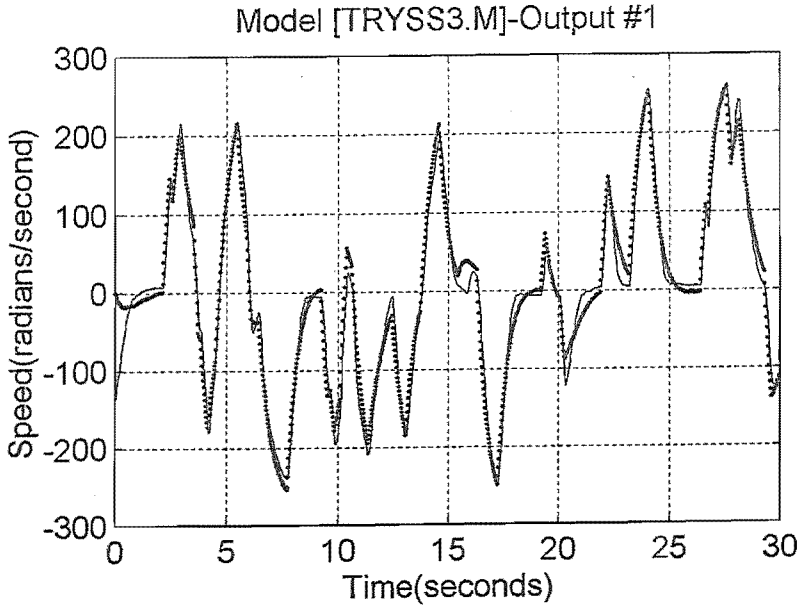
NOTES:

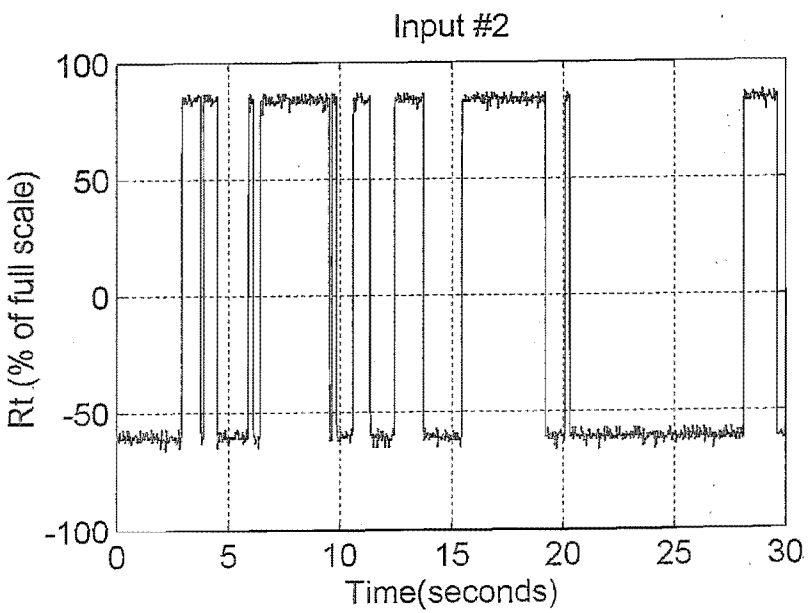
The graphs in this section were produced by using the standard Matlab graphing functions. To test the DCMC models the data was processed with M-files which produced simulations of the calculated state space model plotted against the experimental results.

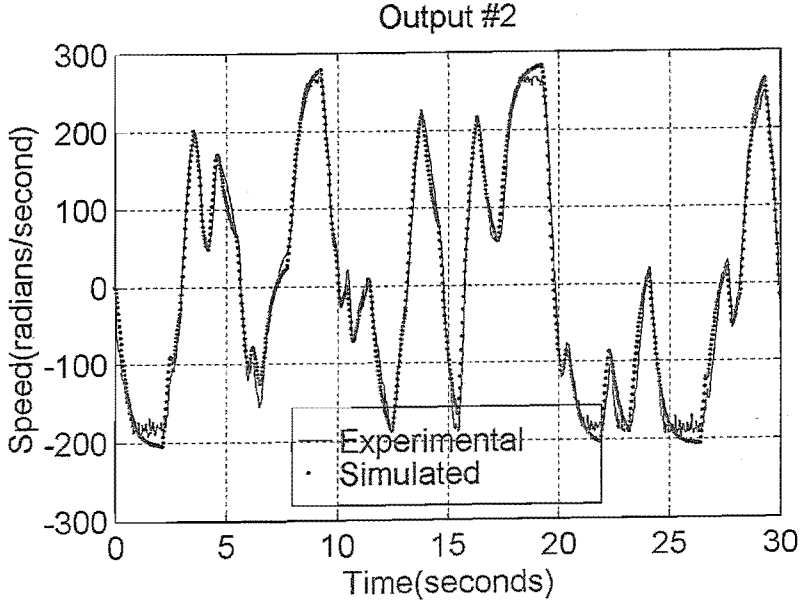
To test the performance of models 2-I and 2-II, the experimental data was processed using TRYSS2.M and TRYSS3.M, respectively.

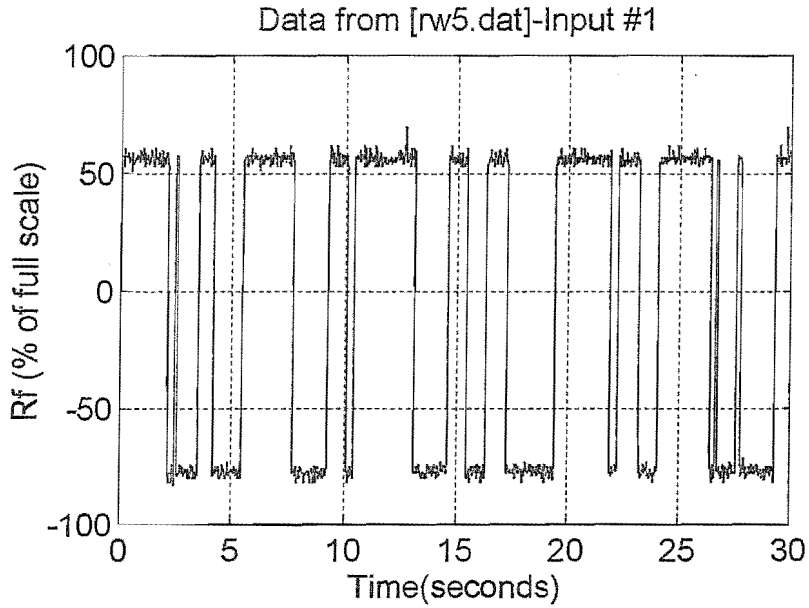
The M-file which was used to process the data is noted on each graph, thus indicating which the of the models it represents.

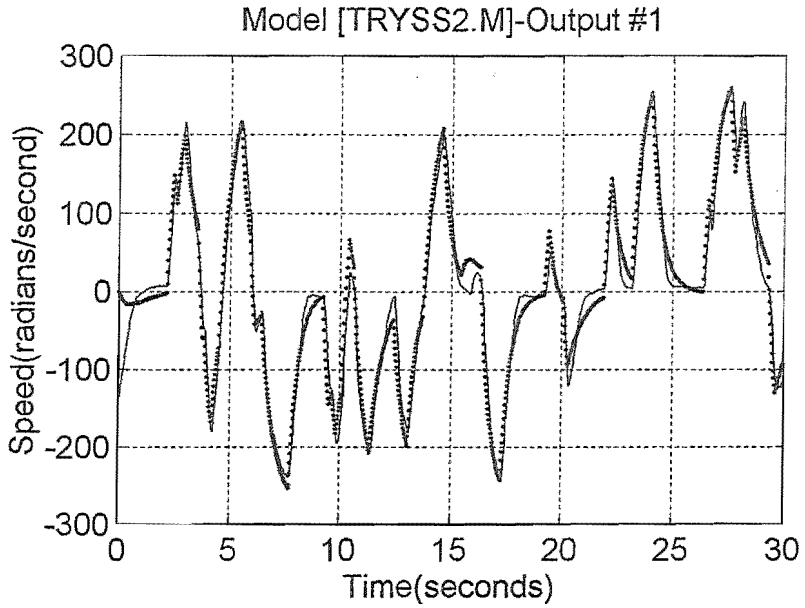

Data from [rw5.dat]-Input #1
Rf (% of full scale)
Time(seconds)

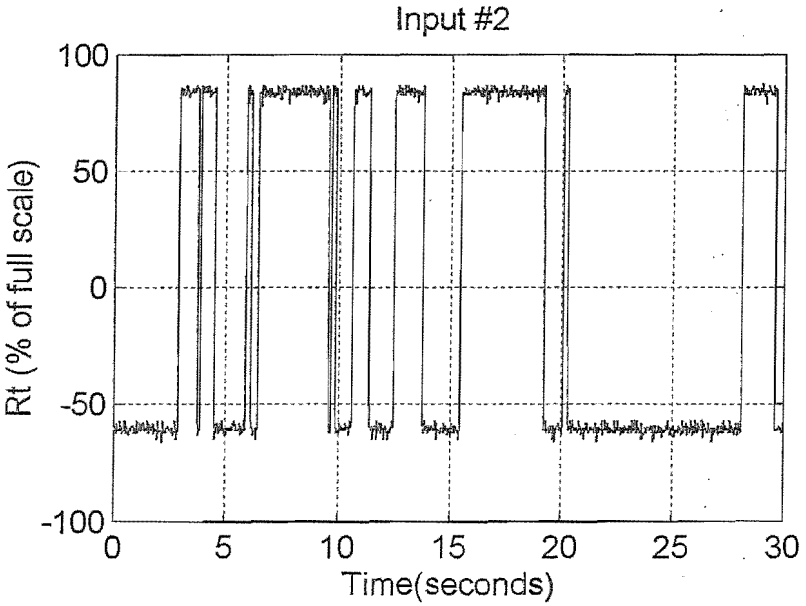

Model [TRYSS3.M]-Output #1
Speed(radians/second)
Time(seconds)

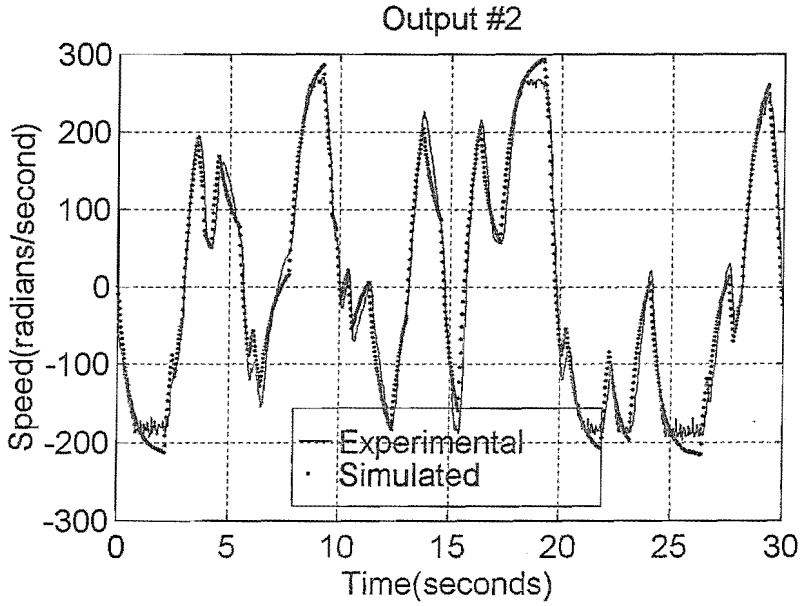

Input #2
Rt (% of full scale)
Time(seconds)

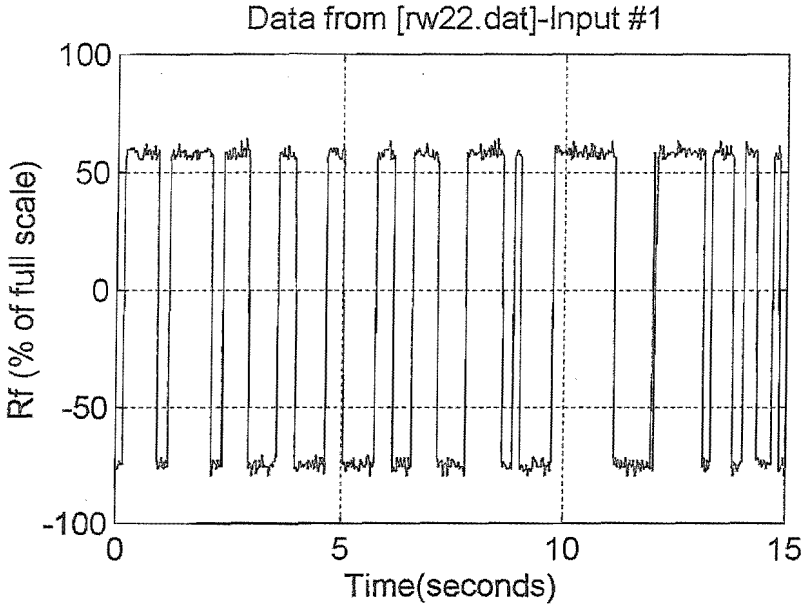

Output #2
Speed(radians/second)
Experimental
Simulated
Time(seconds)

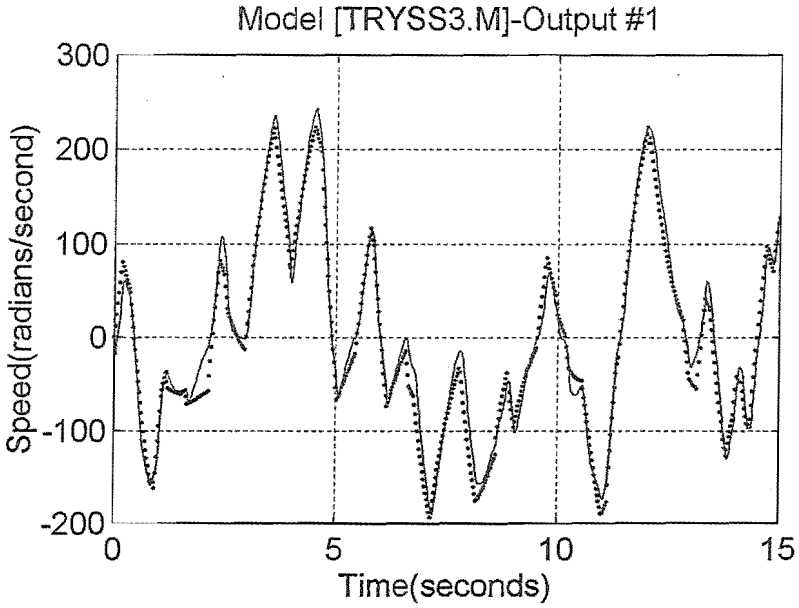

Data from [rw5.dat]-Input #1
Rf (% of full scale)
100
50
0
-50
-100
0
5
10
15
20
25
30
Time(seconds)

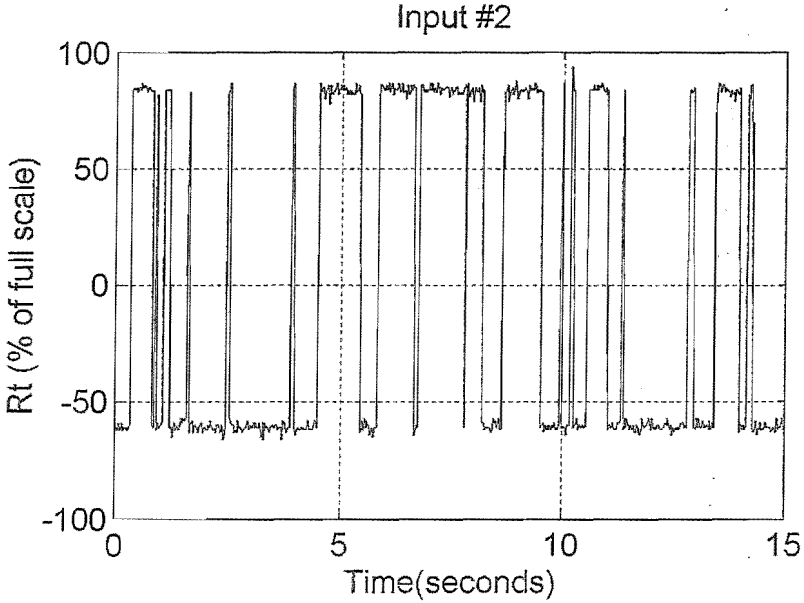

Model [TRYSS2.M]-Output #1
Speed(radians/second)
300
200
100
0
-100
-200
-300
0
5
10
15
20
25
30
Time(seconds)

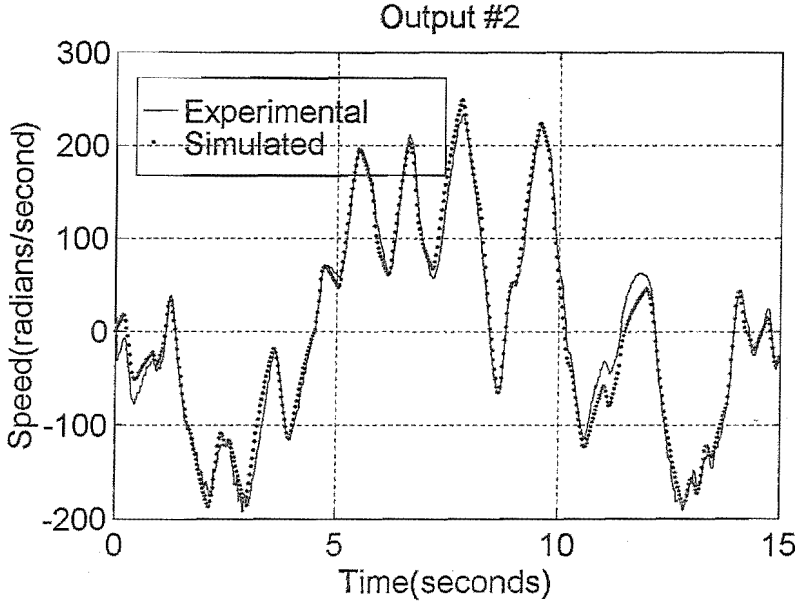

Input #2
Rt (% of full scale)
100
50
0
-50
-100
0
5
10
15
20
25
30
Time(seconds)

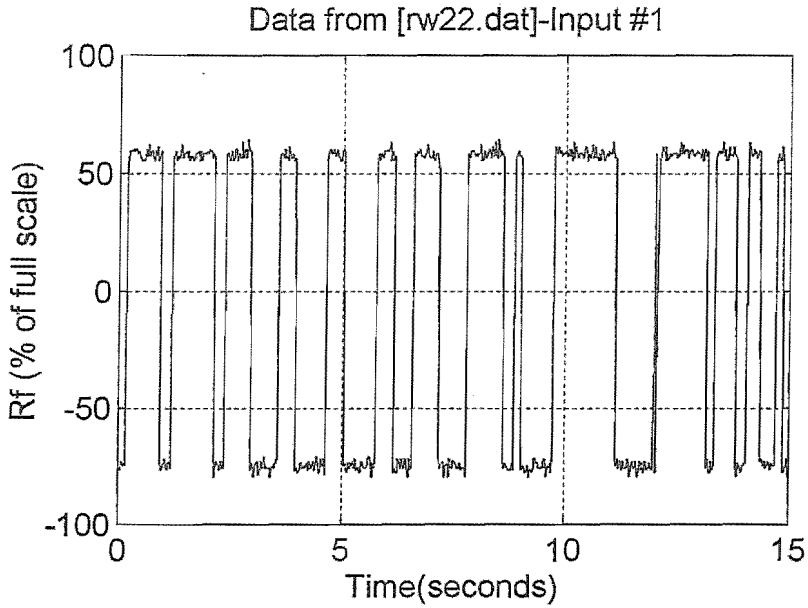

Output #2
Speed(radians/second)
300
200
100
0
-100
-200
-300
Experimental
Simulated
0
5
10
15
20
25
30
Time(seconds)

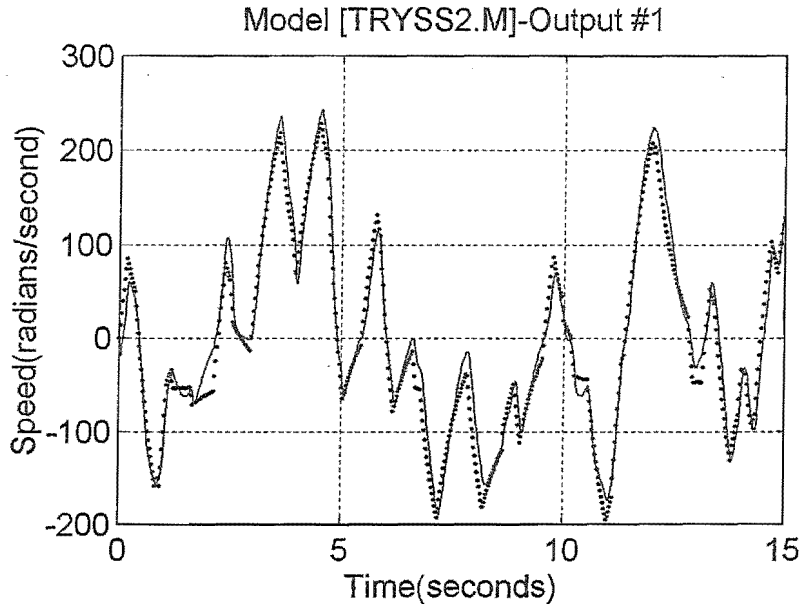

Data from [rw22.dat]-Input #1
Rf (% of full scale)
100
50
0
-50
-100
0
5
10
15
Time(seconds)

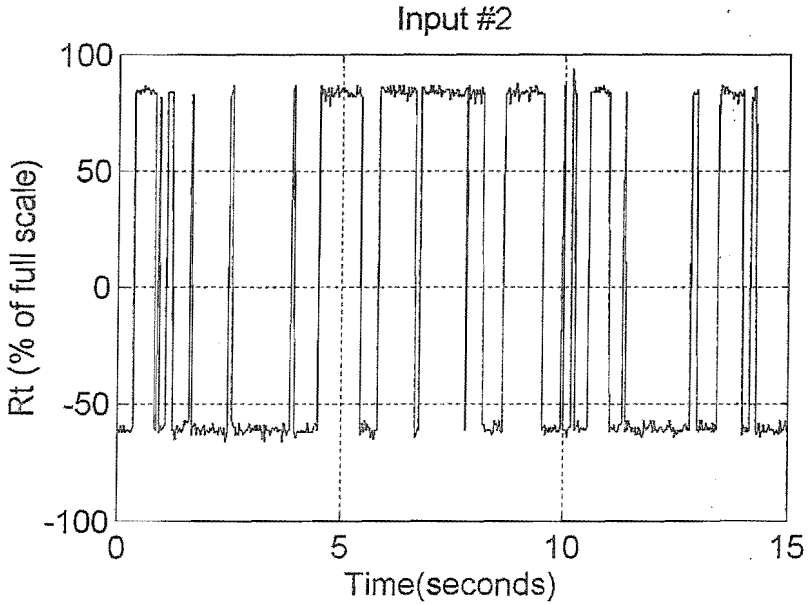

Model [TRYSS3.M]-Output #1
Speed(radians/second)
300
200
100
0
-100
-200
0
5
10
15
Time(seconds)

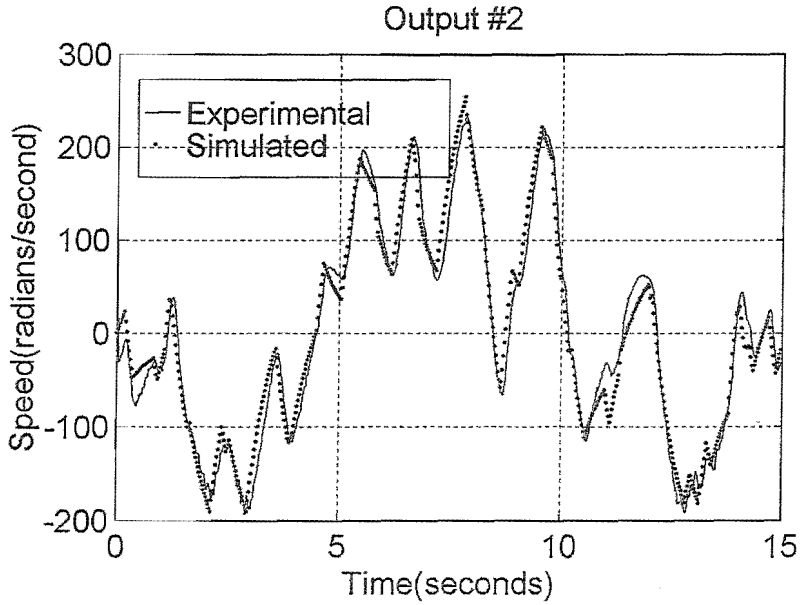

Input #2
Rt (% of full scale)
100
50
0
-50
-100
0
5
10
15
Time(seconds)

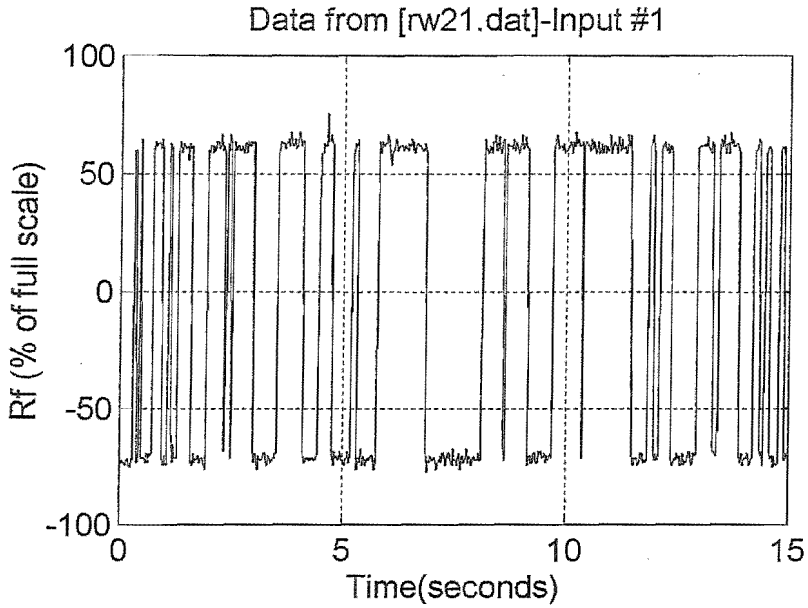

Output #2
Experimental
Simulated
Speed(radians/second)
300
200
100
0
-100
-200
0
5
10
15
Time(seconds)

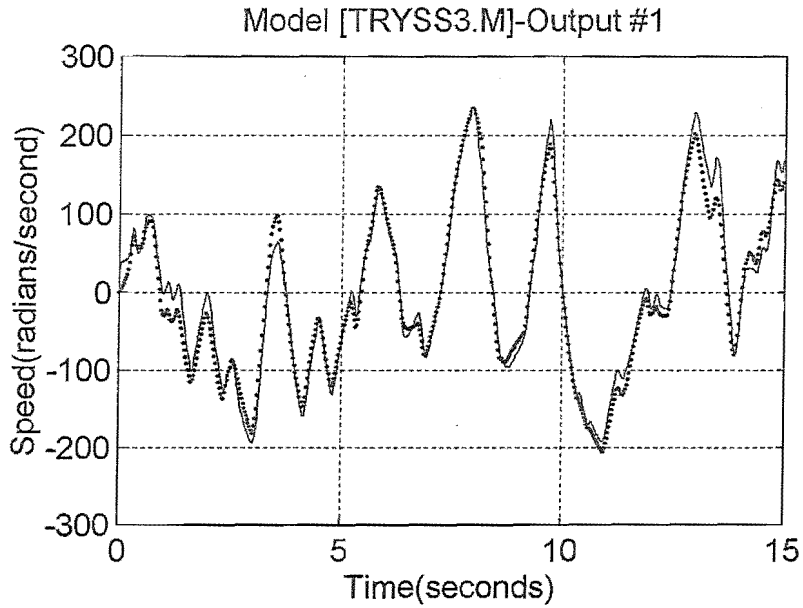

Data from [rw22.dat]-Input #1
Rf (% of full scale)
100
50
0
-50
-100
0
5
10
15
Time(seconds)

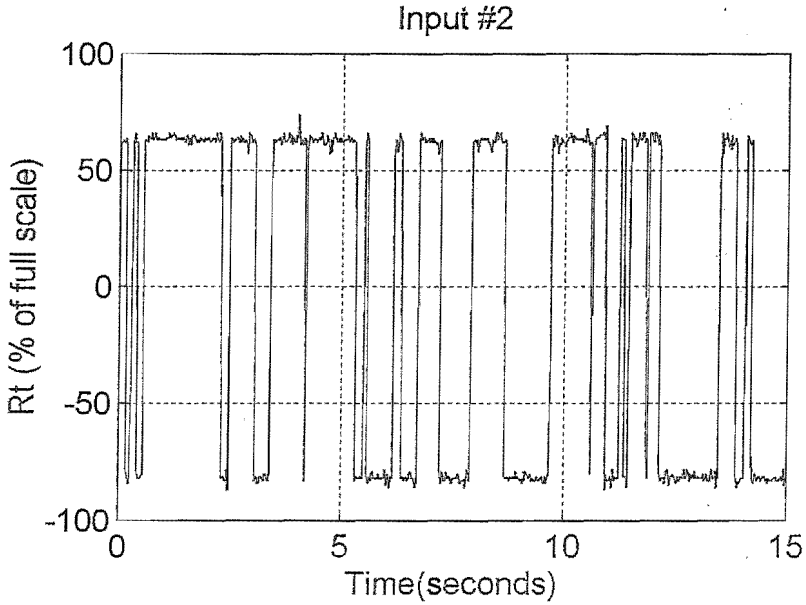

Model [TRYSS2.M]-Output #1
Speed(radians/second)
300
200
100
0
-100
-200
0
5
10
15
Time(seconds)

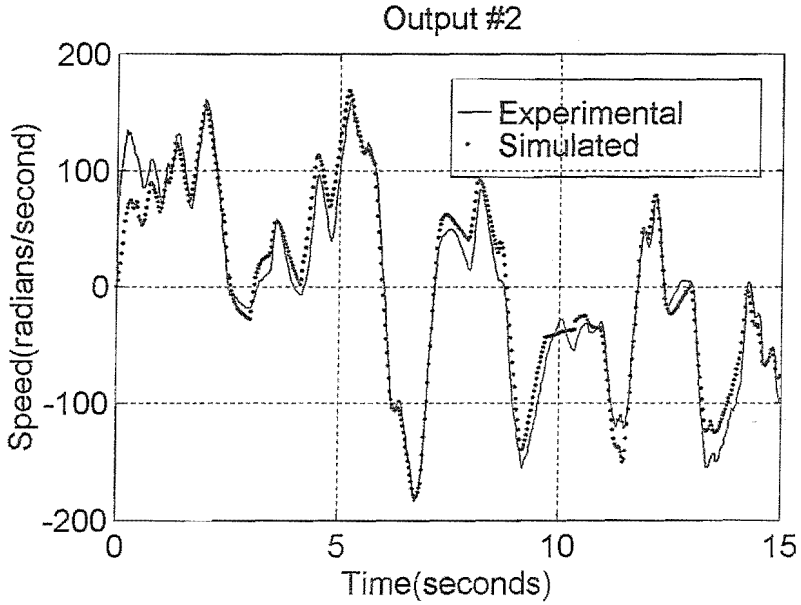

Input #2
Rt (% of full scale)
100
50
0
-50
-100
0
5
10
15
Time(seconds)

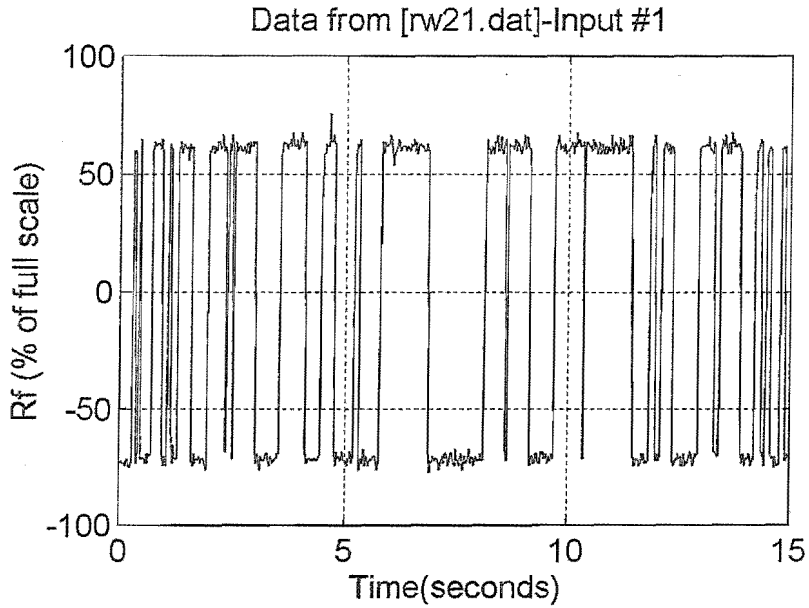

Output #2
Experimental
Simulated
Speed(radians/second)
300
200
100
0
-100
-200
0
5
10
15
Time(seconds)

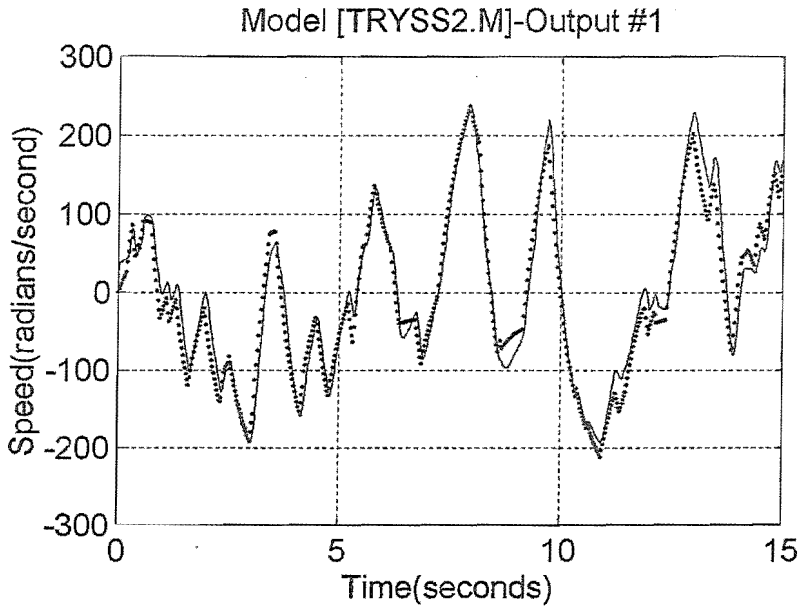

Data from [rw21.dat]-Input #1
Rf (% of full scale)
100
50
0
-50
-100
0
5
10
15
Time(seconds)

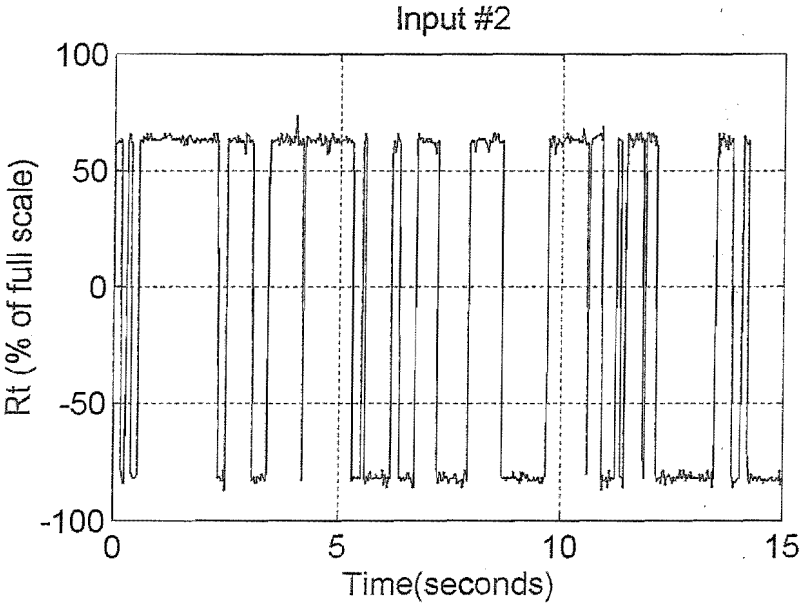

Model [TRYSS3.M]-Output #1
Speed(radians/second)
300
200
100
0
-100
-200
-300
0
5
10
15
Time(seconds)

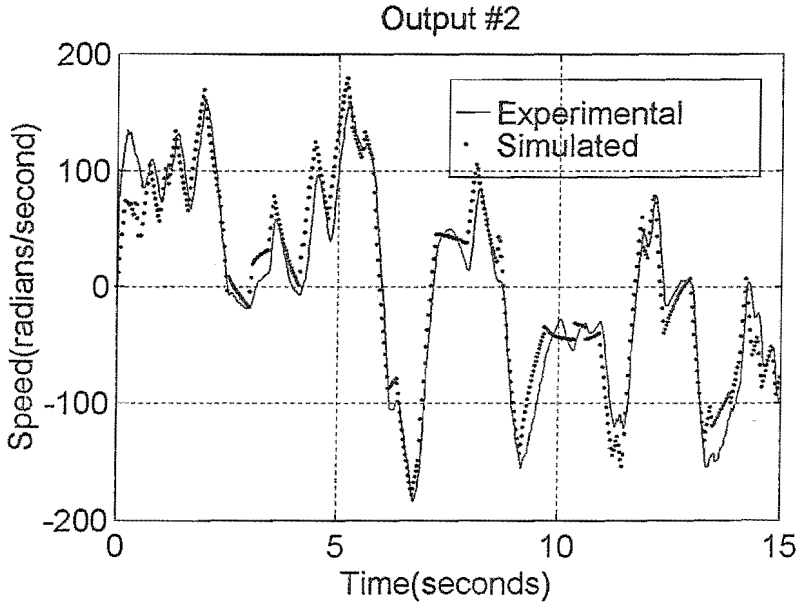

Input #2
Rt (% of full scale)
100
50
0
-50
-100
0
5
10
15
Time(seconds)

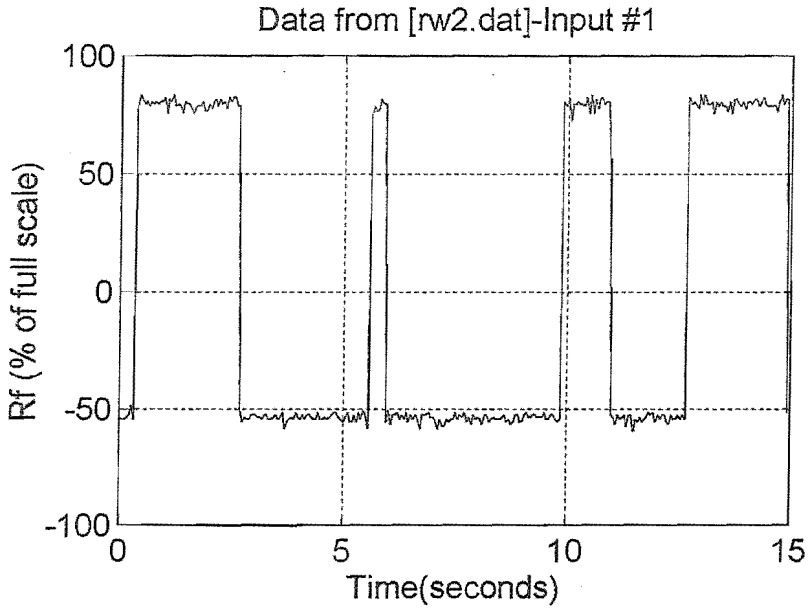

Output #2
Speed(radians/second)
200
100
0
-100
-200
0
5
10
15
Time(seconds)
Experimental
Simulated

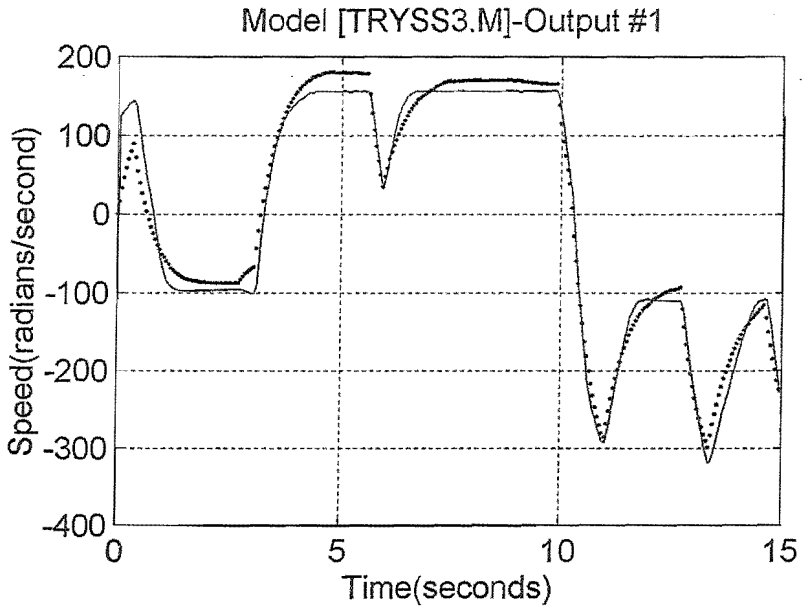

Data from [rw21.dat]-Input #1
Rf (% of full scale)
100
50
0
-50
-100
0
5
10
15
Time(seconds)

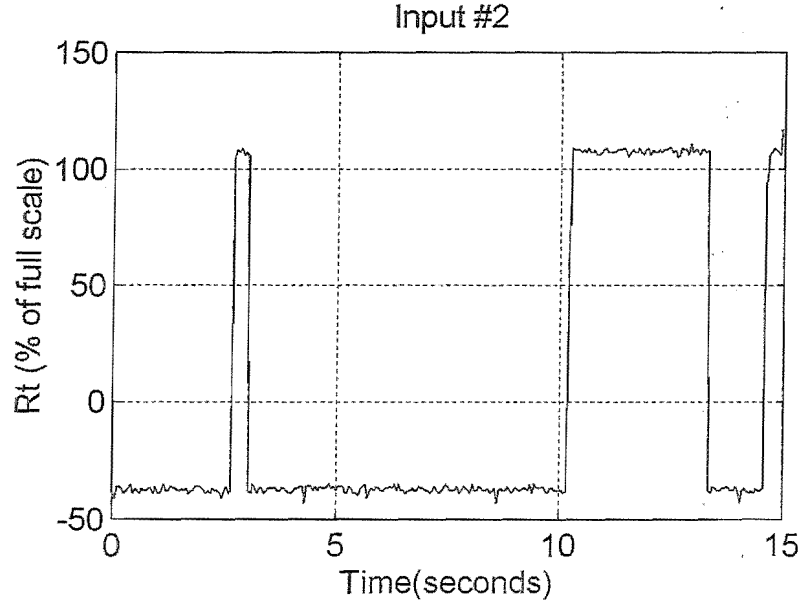

Model [TRYSS2.M]-Output #1
Speed(radians/second)
300
200
100
0
-100
-200
-300
0
5
10
15
Time(seconds)

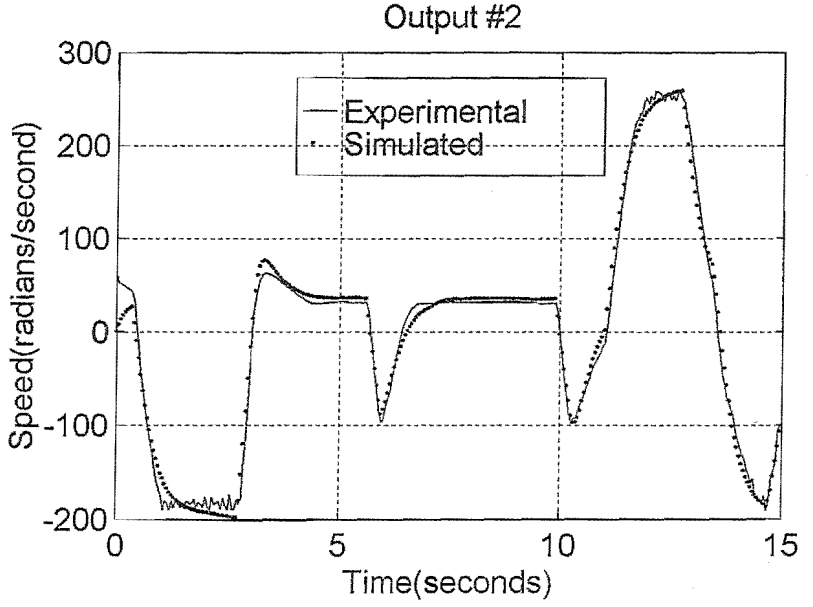

Input #2
Rt (% of full scale)
100
50
0
-50
-100
0
5
10
15
Time(seconds)

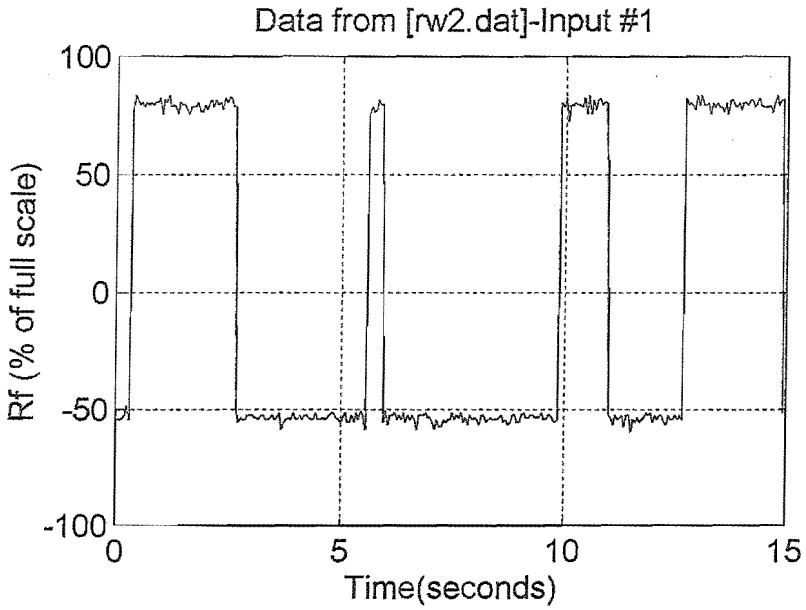

Output #2
Speed(radians/second)
200
100
0
-100
-200
0
5
10
15
Time(seconds)
Experimental
Simulated

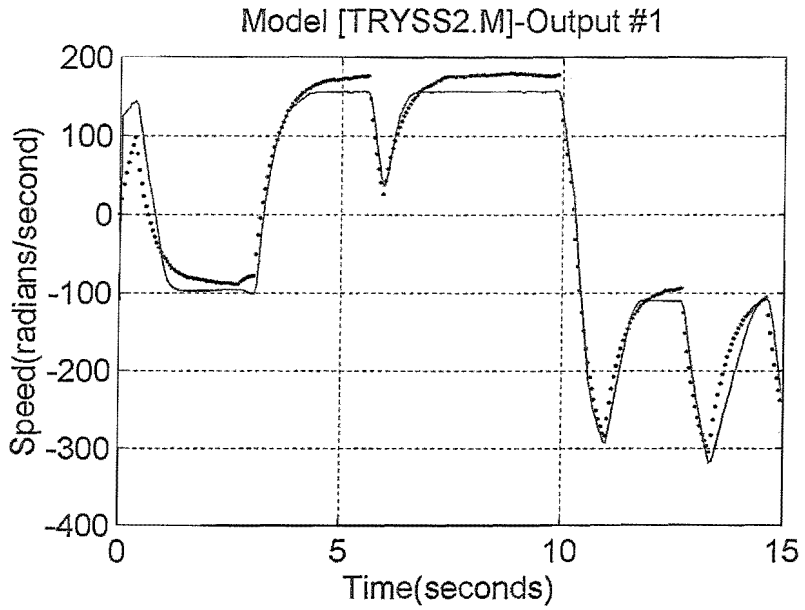

Data from [rw2.dat]-Input #1
Rf (% of full scale)
100
50
0
-50
-100
0
5
10
15
Time(seconds)

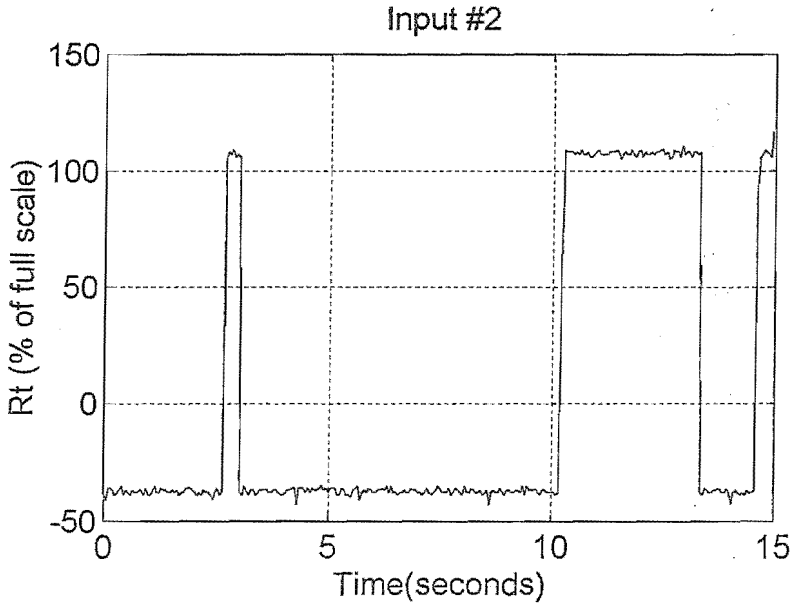

Model [TRYSS3.M]-Output #1
Speed(radians/second)
200
100
0
-100
-200
-300
-400
0
5
10
15
Time(seconds)

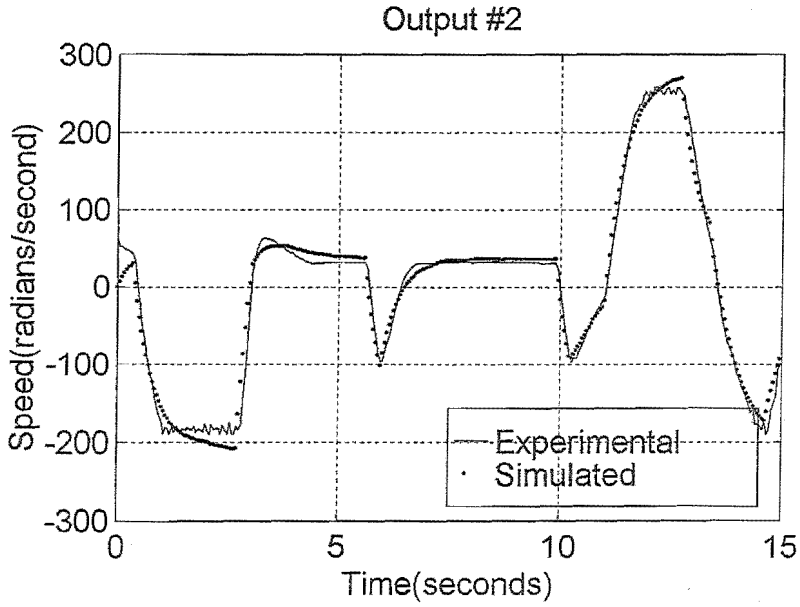

Input #2
Rt (% of full scale)
150
100
50
0
-50
0
5
10
15
Time(seconds)

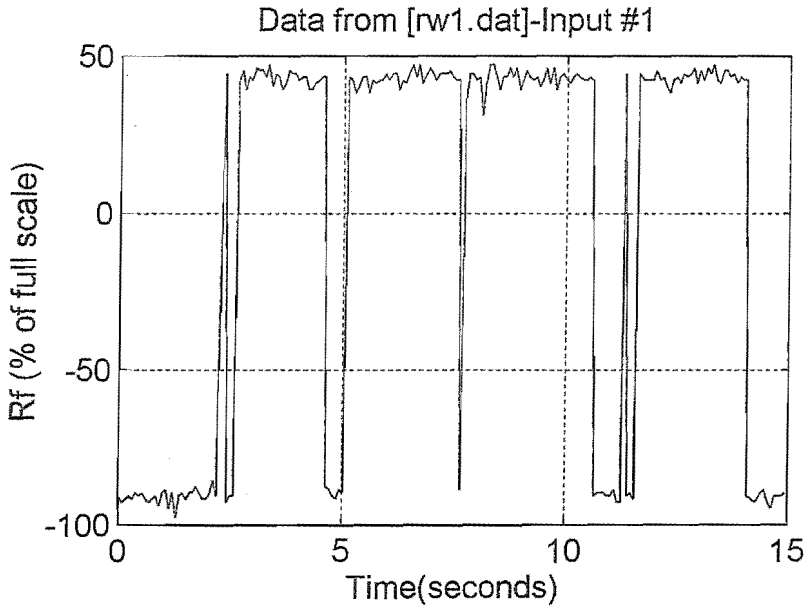

Output #2
Speed(radians/second)
300
200
100
0
-100
-200
Experimental
Simulated
0
5
10
15
Time(seconds)

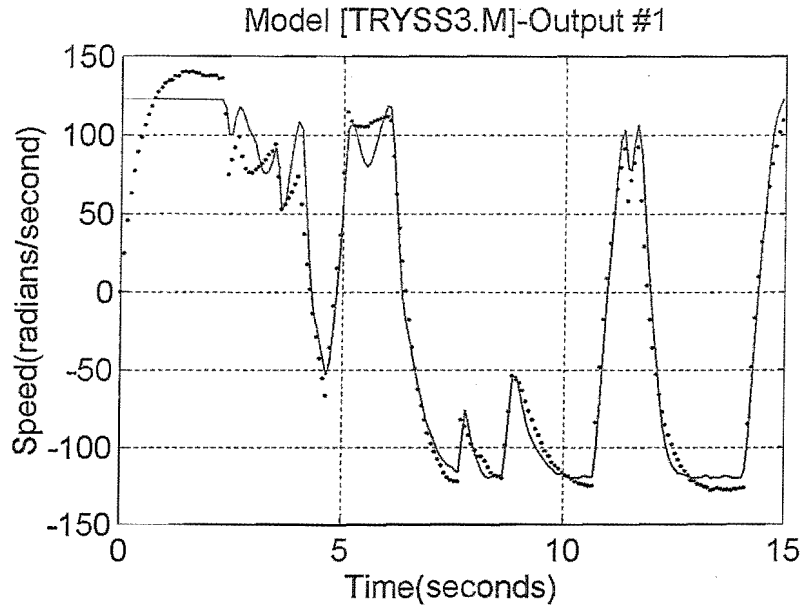

Data from [rw2.dat]-Input #1
Rf (% of full scale)
100
50
0
-50
-100
0
5
10
15
Time(seconds)

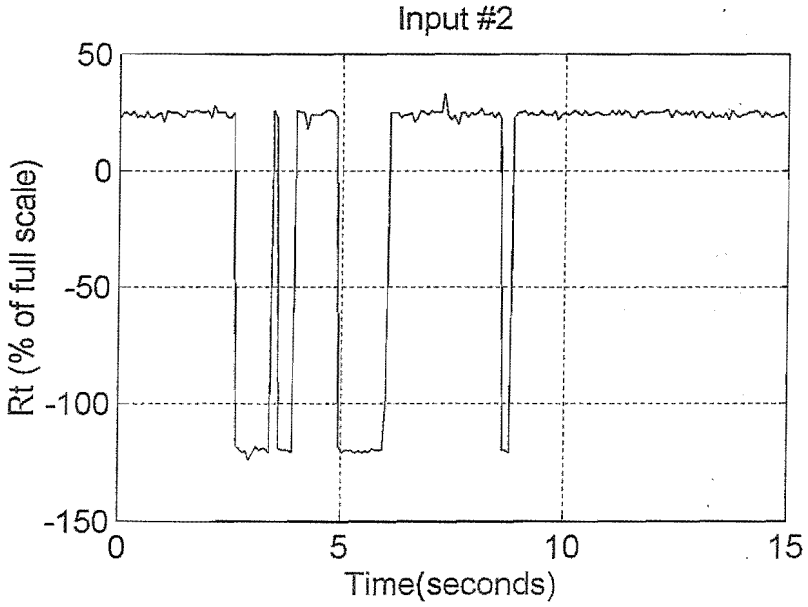

Model [TRYSS2.M]-Output #1
Speed(radians/second)
200
100
0
-100
-200
-300
-400
0
5
10
15
Time(seconds)

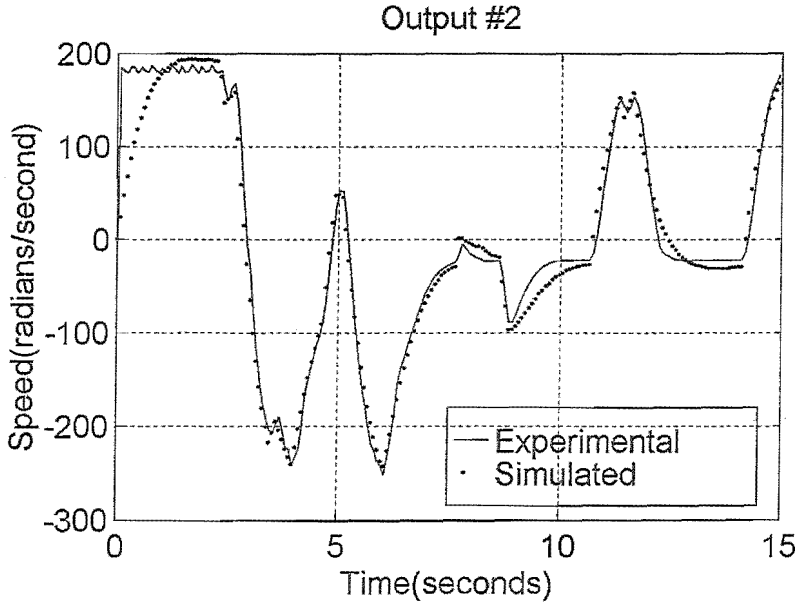

Input #2
Rt (% of full scale)
150
100
50
0
-50
0
5
10
15
Time(seconds)

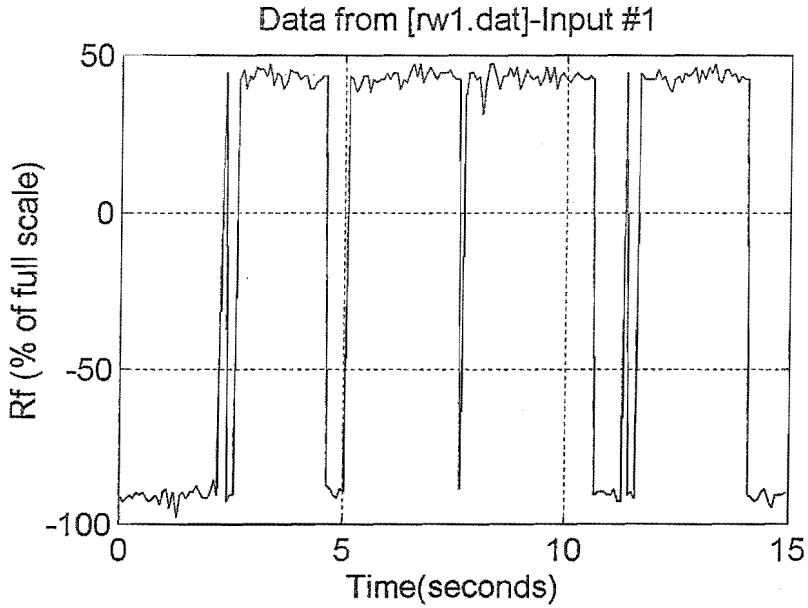

Output #2
Speed(radians/second)
300
200
100
0
-100
-200
-300
0
5
10
15
Time(seconds)
Experimental
Simulated

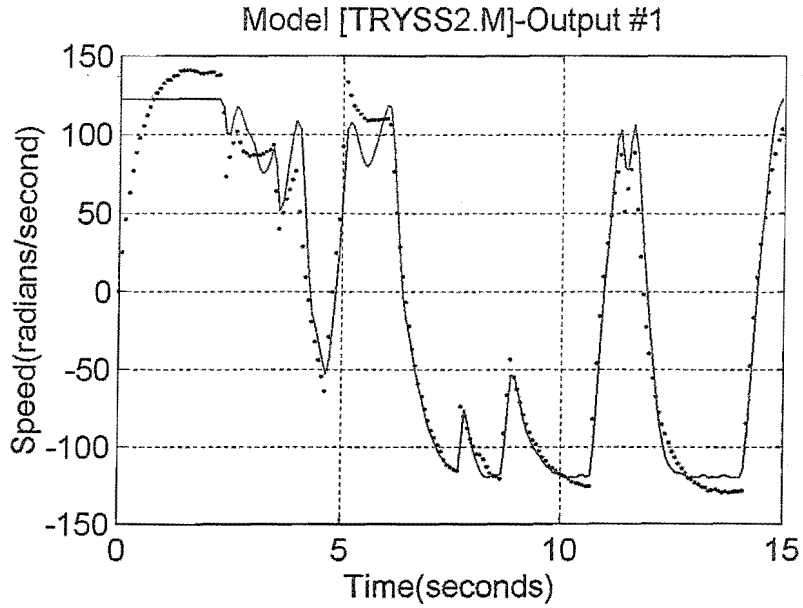

Data from [rw1.dat]-Input #1
Rf (% of full scale)
50
0
-50
-100
0
5
10
15
Time(seconds)

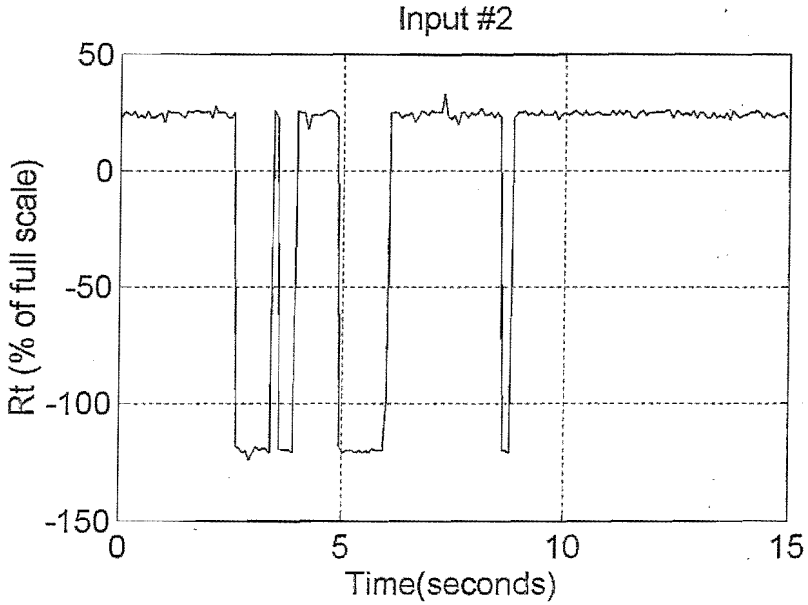

Model [TRYSS3.M]-Output #1
Speed(radians/second)
150
100
50
0
-50
-100
-150
0
5
10
15
Time(seconds)

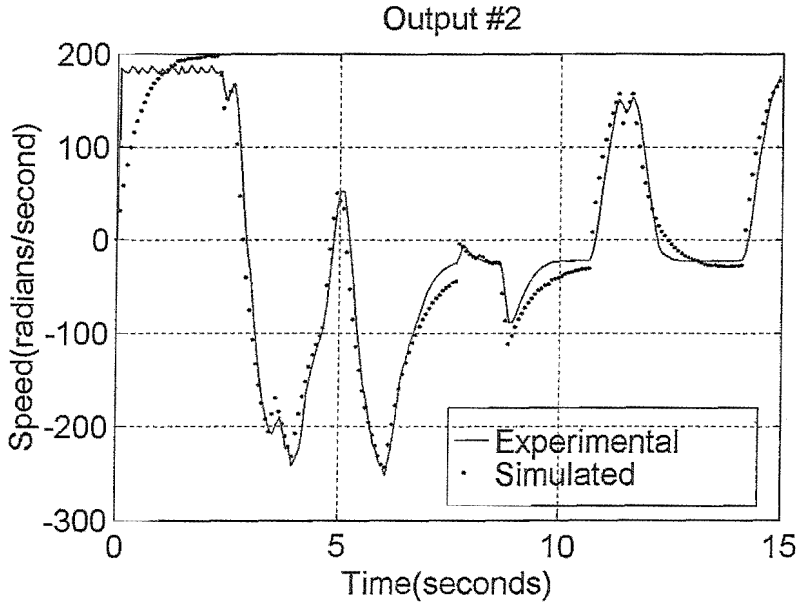

Input #2
Rt (% of full scale)
50
0
-50
-100
-150
0
5
10
15
Time(seconds)

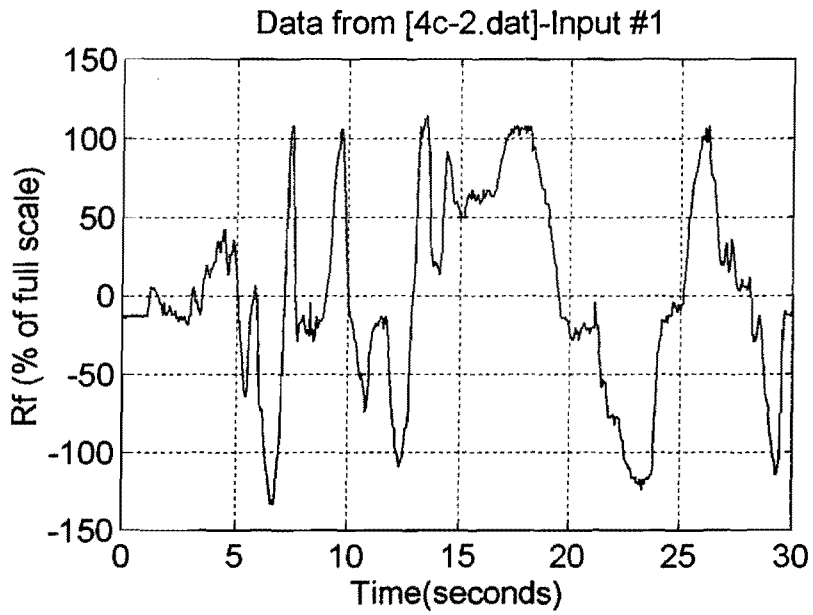

Output #2
Speed(radians/second)
200
100
0
-100
-200
-300
0
5
10
15
Time(seconds)
Experimental
Simulated

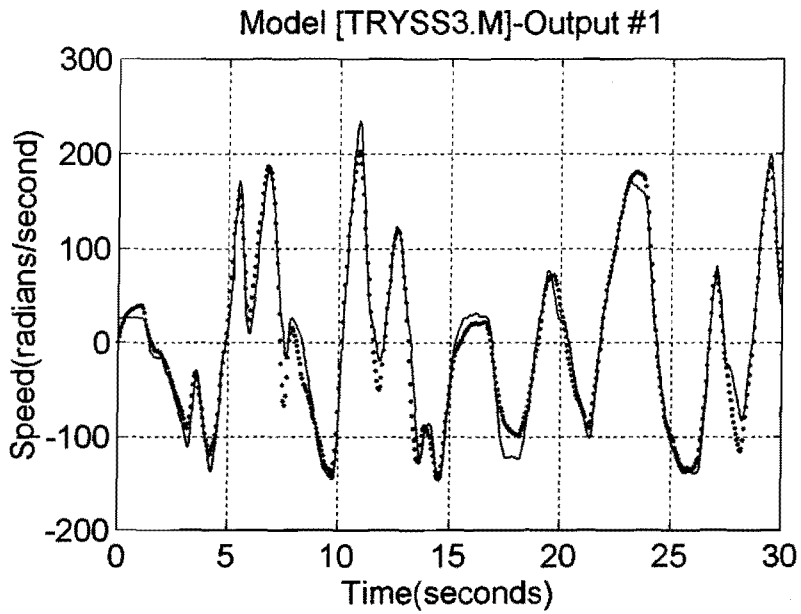

Data from [rw1.dat]-Input #1
Rf (% of full scale)
50
0
-50
-100
0
5
10
15
Time(seconds)

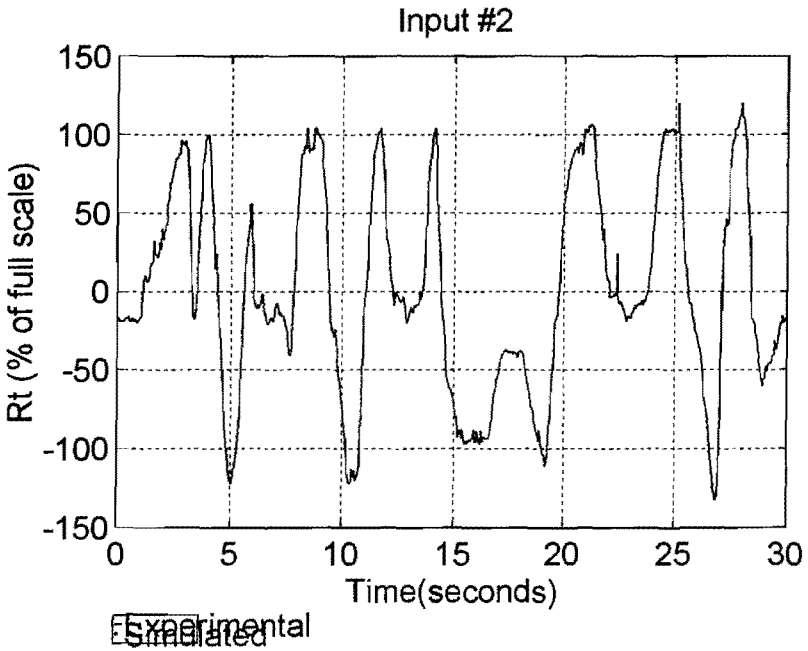

Model [TRYSS2.M]-Output #1
Speed(radians/second)
150
100
50
0
-50
-100
-150
0
5
10
15
Time(seconds)

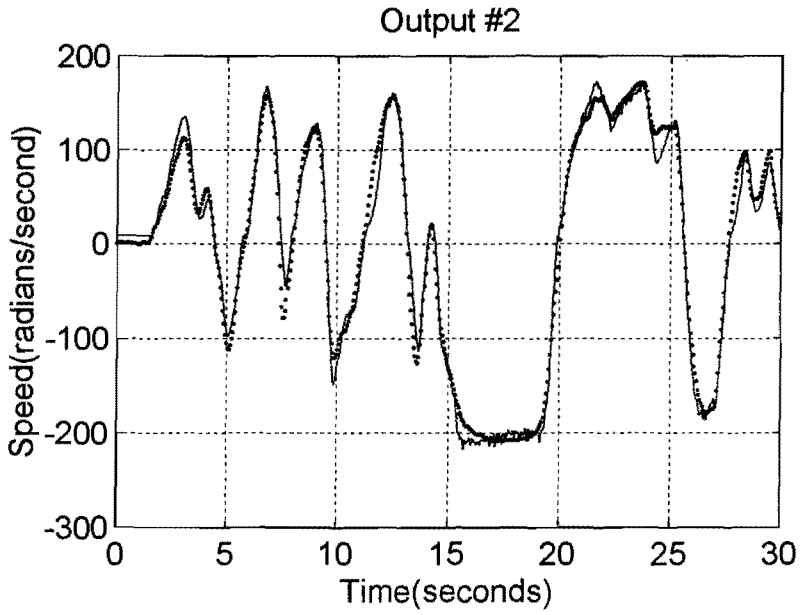

Input #2
Rt (% of full scale)
50
0
-50
-100
-150
0
5
10
15
Time(seconds)

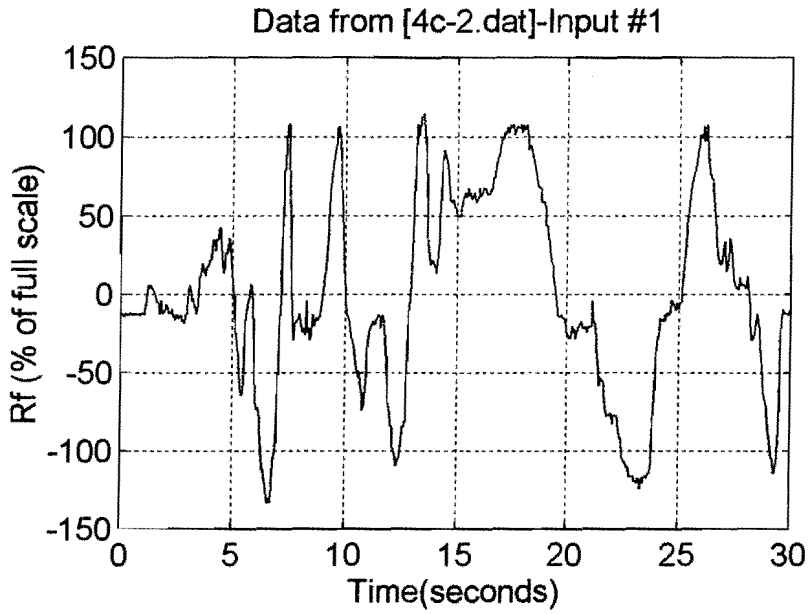

Output #2
Speed(radians/second)
200
100
0
-100
-200
-300
0
5
10
15
Time(seconds)
Experimental
Simulated

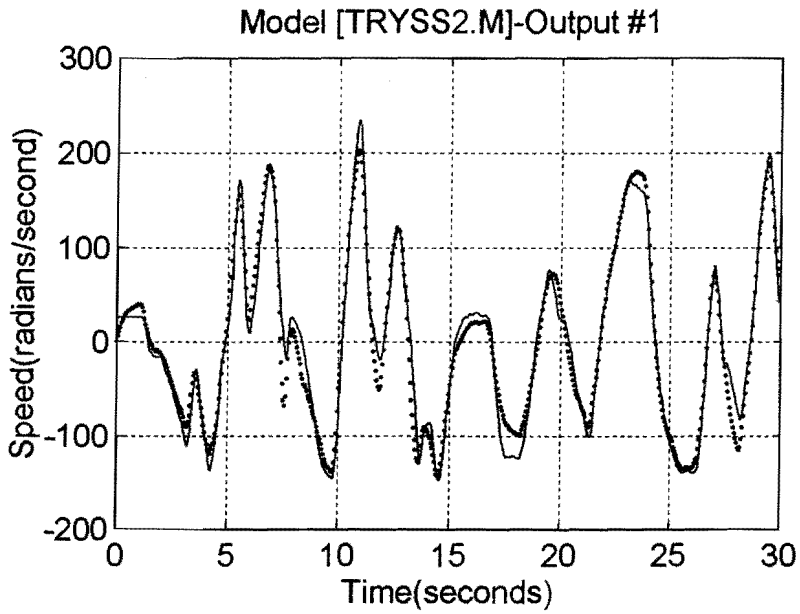

Data from [4c-2.dat]-Input #1
Rf (% of full scale)
Time(seconds)

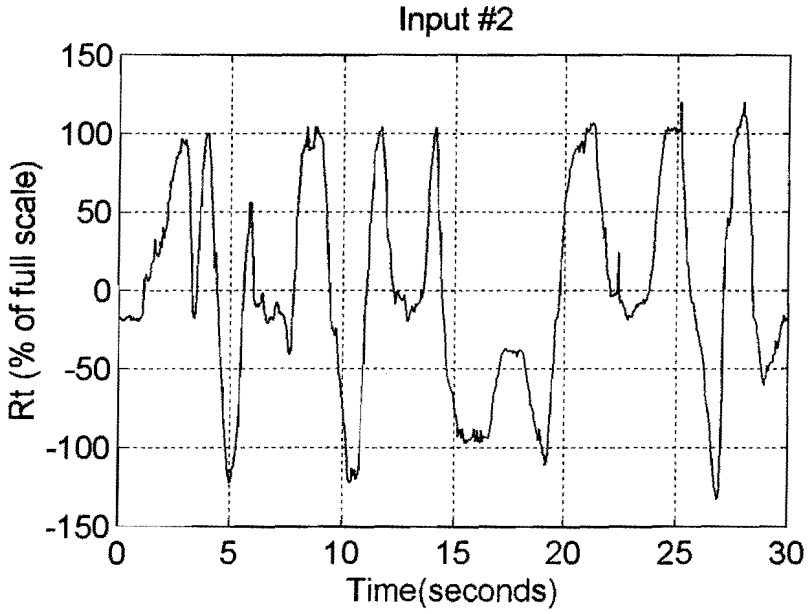

Model [TRYSS3.M]-Output #1
Speed(radians/second)
Time(seconds)

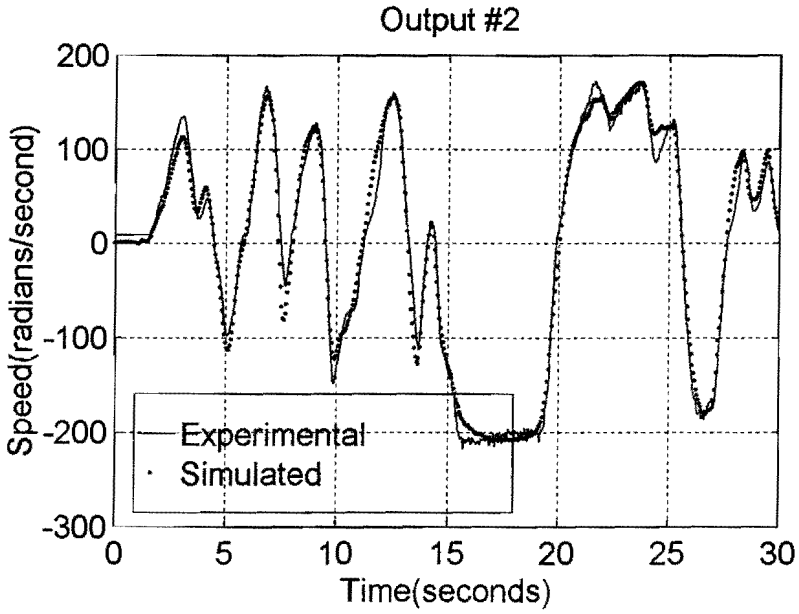

Input #2
Rt (% of full scale)
Time(seconds)
Experimental
Simulated

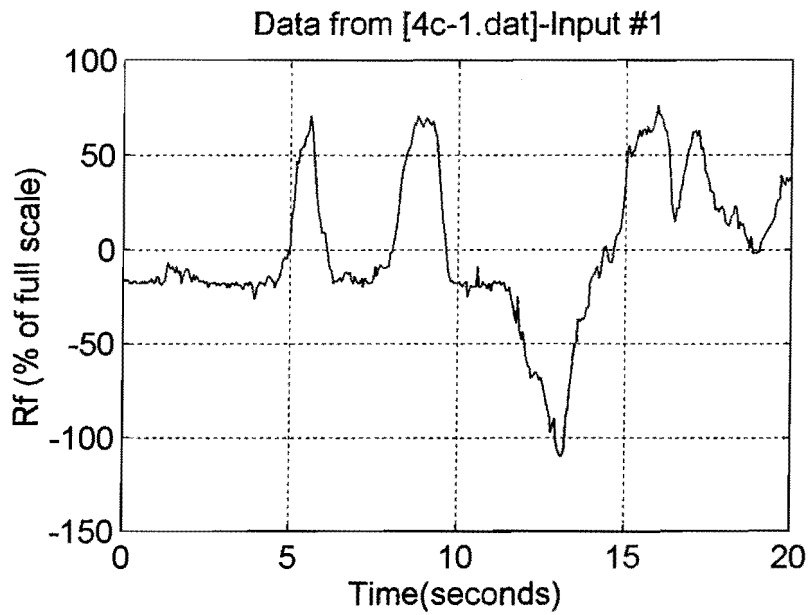

Output #2
Speed(radians/second)
Time(seconds)

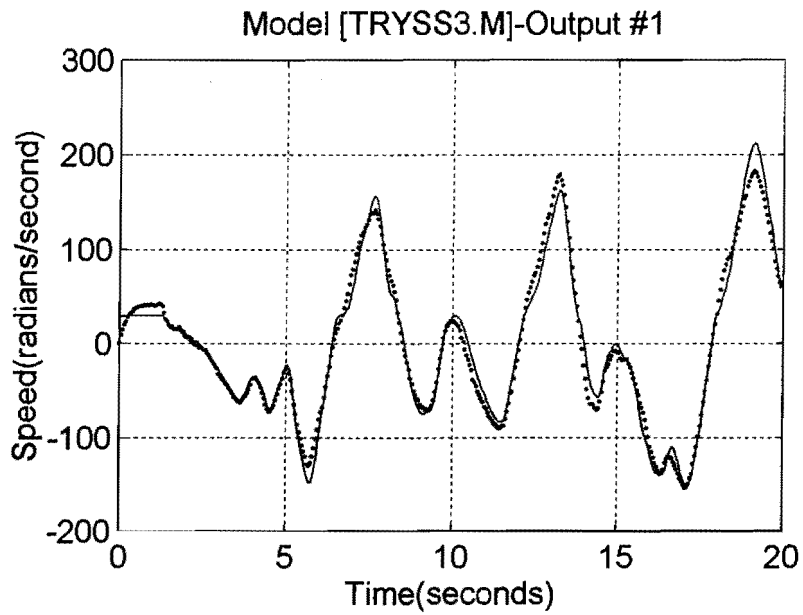

Data from [4c-2.dat]-Input #1
Rf (% of full scale)
Time(seconds)

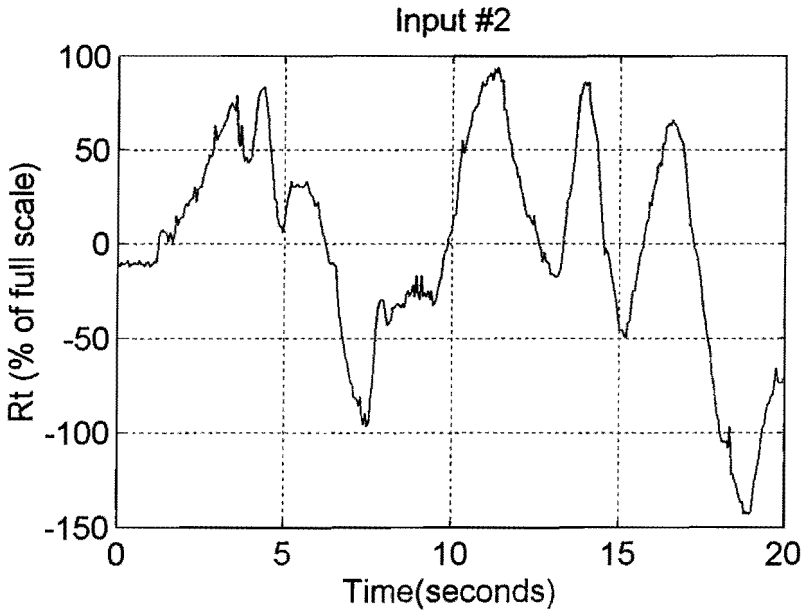

Model [TRYSS2.M]-Output #1
Speed(radians/second)
Time(seconds)

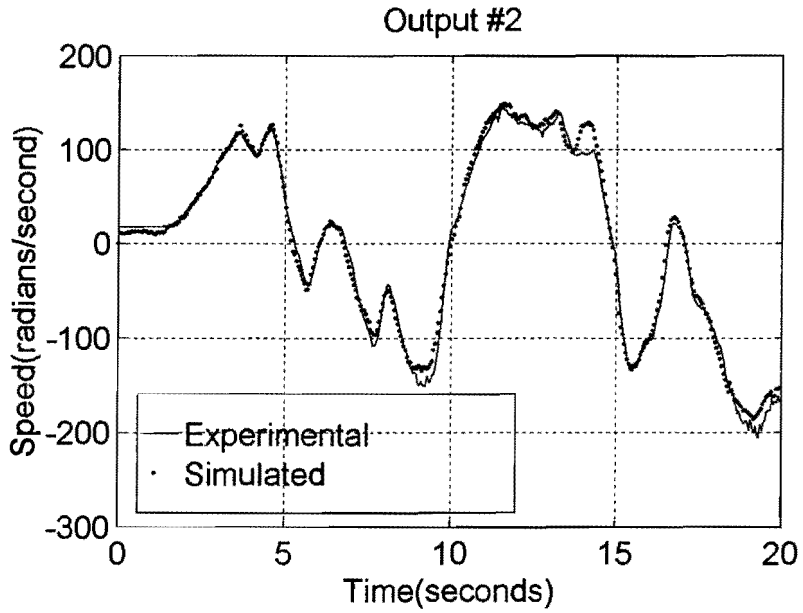

Input #2
Rt (% of full scale)
Time(seconds)

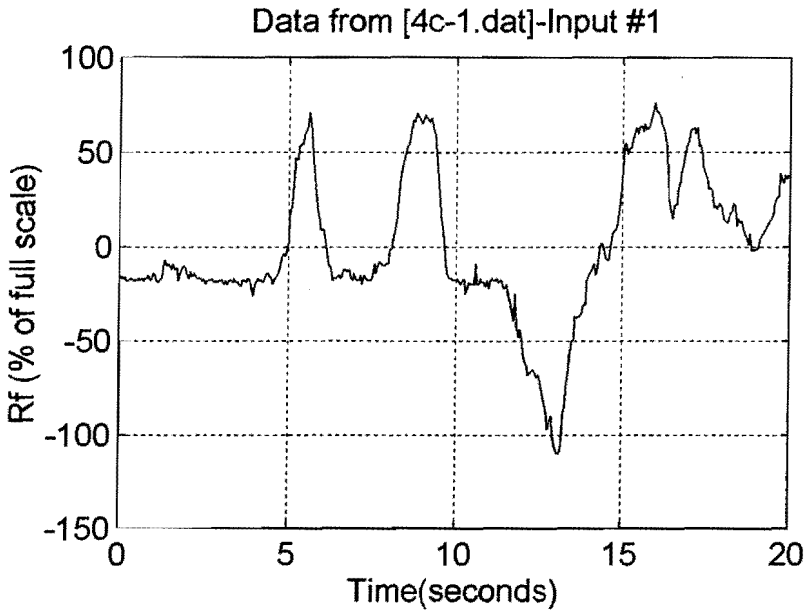

Output #2
Speed(radians/second)
Experimental
Simulated
Time(seconds)

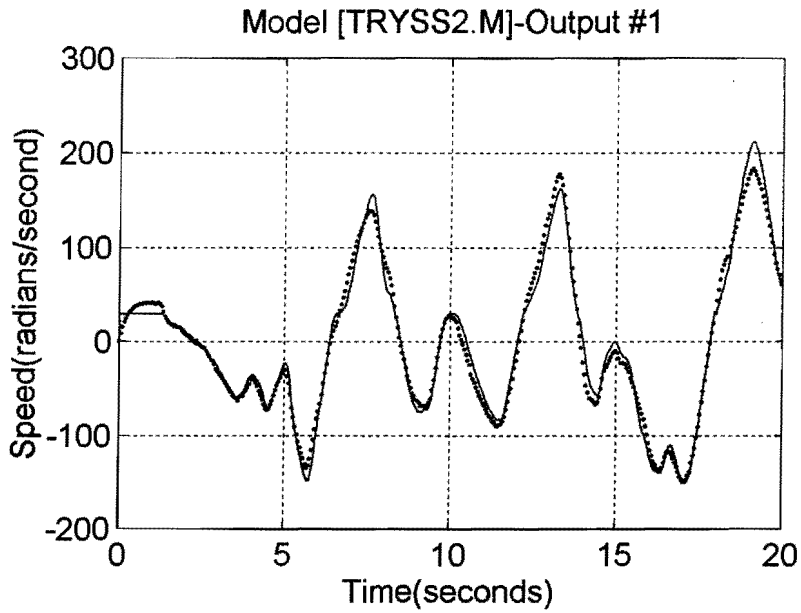

Data from [4c-1.dat]-Input #1
Rf (% of full scale)
Time(seconds)
100
50
0
-50
-100
-150
0
5
10
15
20

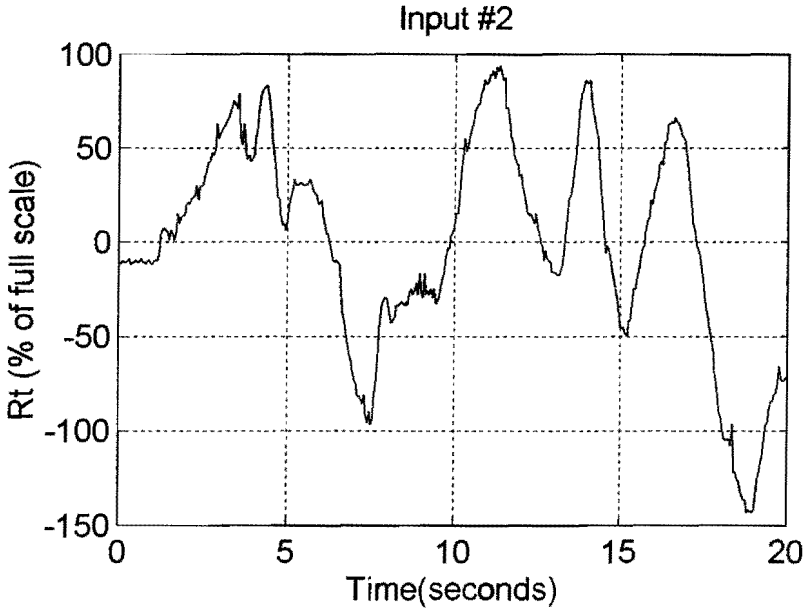

Model [TRYSS3.M]-Output #1
Speed(radians/second)
Time(seconds)
300
200
100
0
-100
-200
0
5
10
15
20

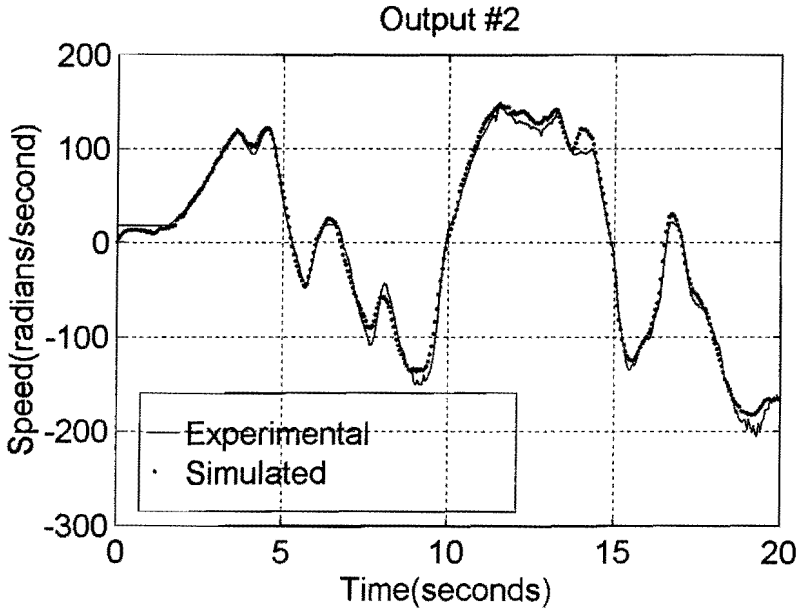

Input #2
Rt (% of full scale)
Time(seconds)
100
50
0
-50
-100
-150
0
5
10
15
20


Output #2
Speed(radians/second)
Time(seconds)
200
100
0
-100
-200
-300
0
5
10
15
20
Experimental
Simulated


Data from [4c-1.dat]-Input #1
Rf (% of full scale)
Time(seconds)
100
50
0
-50
-100
-150
0
5
10
15
20


Model [TRYSS2.M]-Output #1
Speed(radians/second)
Time(seconds)
300
200
100
0
-100
-200
0
5
10
15
20


Input #2
Rt (% of full scale)
Time(seconds)
100
50
0
-50
-100
-150
0
5
10
15
20


Output #2
Speed(radians/second)
Time(seconds)
Experimental
Simulated
200
100
0
-100
-200
-300
0
5
10
15
20